Table of Contents

> From: bjpcjp/scikit-and-tensorflow-workbooks



based on "Hands-On Machine Learning with Scikit-Learn & TensorFlow" (O'Reilly, Aurelien Geron)

book chapters

1) Intro to Machine Learning 2) Example end-to-end Machine Learning project (California Housing dataset) 3) Basic Classification 4) Training Techniques 5) Support Vector Machines 6) Decision Trees 7) Ensemble Learning & Random Forests 8) Dimensionality Reduction 9) TensorFlow Installation & Checkout 10) TensorFlow & Neural Nets 11) TensorFlow Training 12) TensorFlow on Distributed Hardware 13) Convolutional Neural Nets 14) Recurrent Neural Nets 15) Autoencoders 16) Reinforcement Learning

Get Dataset & Create Workspace

```
import os
import tarfile
from six.moves import urllib

import pandas as pd
import numpy  as np
```

```
DOWNLOAD_ROOT = "https://raw.githubusercontent.com/ageron/handso
n-ml/master/"
HOUSING_PATH = "datasets/housing"
HOUSING_URL = DOWNLOAD_ROOT + HOUSING_PATH + "/housing.tgz"
```

```
def fetch_housing_data(
    housing_url=HOUSING_URL,
    housing_path=HOUSING_PATH):

    # create datasets/housing directory if needed
    if not os.path.isdir(housing_path):
        os.makedirs(housing_path)

    tgz_path = os.path.join(housing_path, "housing.tgz")

    # retrieve tarfile
    urllib.request.urlretrieve(housing_url, tgz_path)

    # extract tarfile & close path
    housing_tgz = tarfile.open(tgz_path)
    housing_tgz.extractall(path=housing_path)
    housing_tgz.close()
```

```
def load_housing_data(
    housing_path=HOUSING_PATH):

    csv_path = os.path.join(housing_path, "housing.csv")
    return pd.read_csv(csv_path)
```

```
# do it
#fetch_housing_data() -- already downloaded - static dataset
housing = load_housing_data()
```

Data structure - quick peek

```
housing.head()
```

	longitude	latitude	housing_median_age	total_rooms	total_be
0	-122.23	37.88	41.0	880.0	129.0
1	-122.22	37.86	21.0	7099.0	1106.0
2	-122.24	37.85	52.0	1467.0	190.0
3	-122.25	37.85	52.0	1274.0	235.0
4	-122.25	37.85	52.0	1627.0	280.0

So... what's in the dataset?

```
# housing is a Pandas DataFrame.
# untouched datafile: 20640 records, 10 cols (9 float, 1 text)
housing.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 20640 entries, 0 to 20639
Data columns (total 10 columns):
longitude             20640 non-null float64
latitude              20640 non-null float64
housing_median_age    20640 non-null float64
total_rooms           20640 non-null float64
total_bedrooms        20433 non-null float64
population            20640 non-null float64
households            20640 non-null float64
median_income         20640 non-null float64
median_house_value    20640 non-null float64
ocean_proximity       20640 non-null object
dtypes: float64(9), object(1)
memory usage: 1.6+ MB
```

```
# let's see if ocean_proximity can be lumped into categories:
housing['ocean_proximity'].value_counts()
```

```
<1H OCEAN     9136
INLAND        6551
NEAR OCEAN    2658
NEAR BAY      2290
ISLAND           5
Name: ocean_proximity, dtype: int64
```

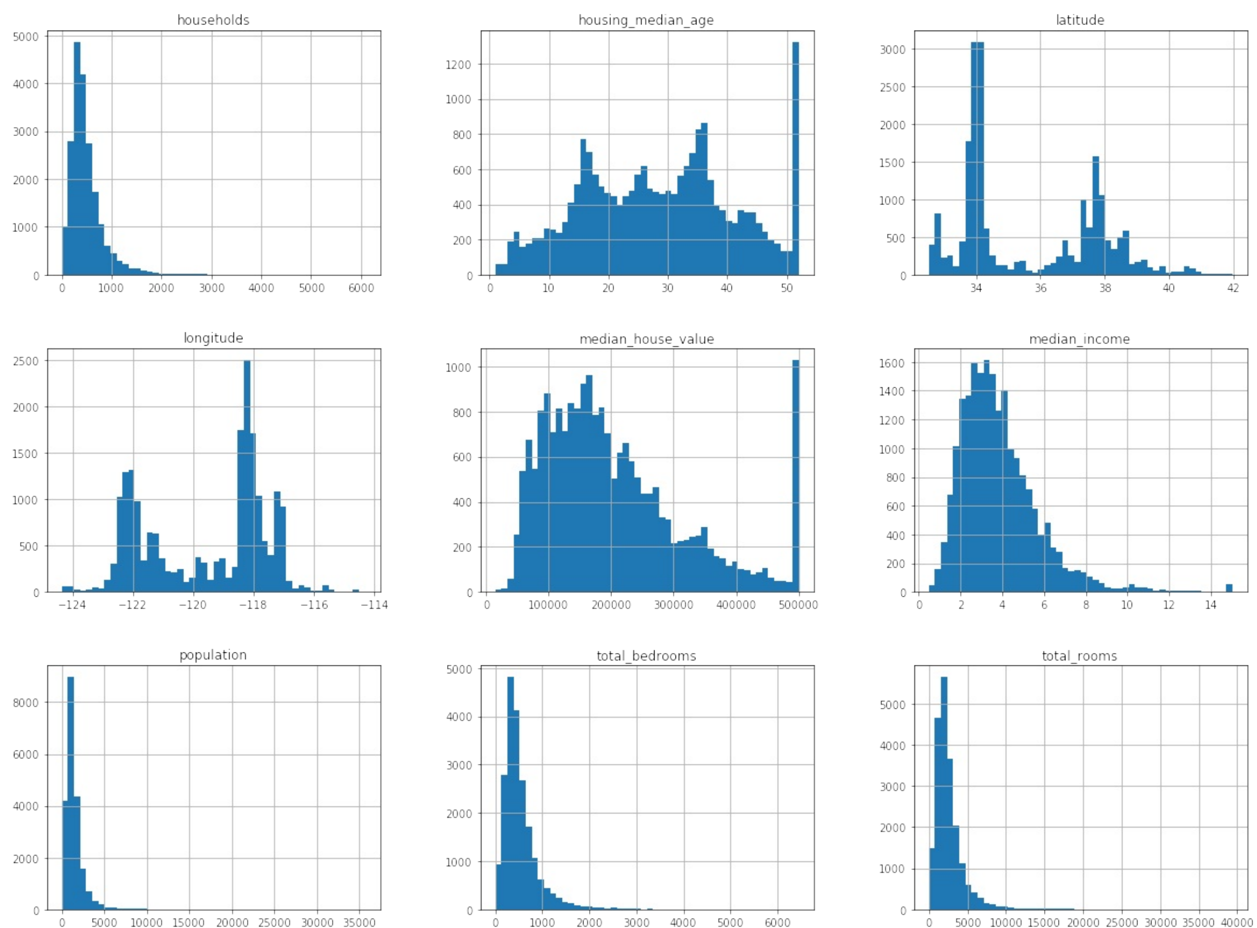
```
# percentiles analysis of each feature
housing.describe()
```

	longitude	latitude	housing_median_age	total_r
count	20640.000000	20640.000000	20640.000000	20640.00
mean	-119.569704	35.631861	28.639486	2635.763
std	2.003532	2.135952	12.585558	2181.615
min	-124.350000	32.540000	1.000000	2.000000
25%	-121.800000	33.930000	18.000000	1447.750
50%	-118.490000	34.260000	29.000000	2127.000
75%	-118.010000	37.710000	37.000000	3148.000
max	-114.310000	41.950000	52.000000	39320.00

```
# feature histograms
%matplotlib inline
import matplotlib.pyplot as plt

housing.hist(bins=50, figsize=(20,15))
```

```
array([[<matplotlib.axes._subplots.AxesSubplot object at 0x7f4a7
92b5438>,
        <matplotlib.axes._subplots.AxesSubplot object at 0x7f4a7
5f1c2e8>,
        <matplotlib.axes._subplots.AxesSubplot object at 0x7f4a7
5f39d68>],
       [<matplotlib.axes._subplots.AxesSubplot object at 0x7f4a7
5eaf7b8>,
        <matplotlib.axes._subplots.AxesSubplot object at 0x7f4a7
5e7acc0>,
        <matplotlib.axes._subplots.AxesSubplot object at 0x7f4a7
5e40438>],
       [<matplotlib.axes._subplots.AxesSubplot object at 0x7f4a7
5e0a860>,
        <matplotlib.axes._subplots.AxesSubplot object at 0x7f4a7
5dce198>,
        <matplotlib.axes._subplots.AxesSubplot object at 0x7f4a7
5d1d1d0>]], dtype=object)
```



Create a test set

```
# split dataset into training (80%) and test (20%) subsets

import numpy as np

def split_train_test(
    data, test_ratio):

    shuffled_indices = np.random.permutation(len(data))

    test_set_size = int(len(data) * test_ratio)

    test_indices = shuffled_indices[:test_set_size]
    train_indices = shuffled_indices[test_set_size:]

    return data.iloc[train_indices], data.iloc[test_indices]
```

```
train_set, test_set = split_train_test(housing, 0.2)
```

```
print(len(train_set), "train +", len(test_set), "test")
```

```
16512 train + 4128 test
```

```
# create method for ensuring consistent test sets across multipl
e runs
# (new test sets won't contain instances in previous training se
ts.)

# example method:
# compute hash of each instance
# keep only the last byte
# include instance in test set if value < 51 (20% of 256)

import hashlib

def test_set_check(
    identifier, test_ratio, hash):

    return hash(np.int64(identifier)).digest()[-1] < 256 * test_
ratio

def split_train_test_by_id(
    data, test_ratio, id_column, hash=hashlib.md5):

    ids = data[id_column]
    in_test_set = ids.apply(
        lambda id_: test_set_check(
            id_, test_ratio, hash))

    return data.loc[~in_test_set], data.loc[in_test_set]
```

```
# housing dataset doesn't have ID attribute,
# so let's add an index to it.

housing_with_id = housing.reset_index()

train_set, test_set = split_train_test_by_id(
    housing_with_id, 0.2, "index")
```

```
train_set.head()
```

	index	longitude	latitude	housing_median_age	total_rooms
0	0	-122.23	37.88	41.0	880.0
1	1	-122.22	37.86	21.0	7099.0
2	2	-122.24	37.85	52.0	1467.0
3	3	-122.25	37.85	52.0	1274.0
6	6	-122.25	37.84	52.0	2535.0

```
test_set.head()
```

	index	longitude	latitude	housing_median_age	total_rooms
4	4	-122.25	37.85	52.0	1627.0
5	5	-122.25	37.85	52.0	919.0
11	11	-122.26	37.85	52.0	3503.0
20	20	-122.27	37.85	40.0	751.0
23	23	-122.27	37.84	52.0	1688.0

```
# a better index:
# let's use longitude & latitude to build stable identifier

housing_with_id["id"] = housing["longitude"] * 1000 + housing["l
atitude"]

train_set, test_set = split_train_test_by_id(
    housing_with_id, 0.2, "id")
```

```
train_set.head()
```

	index	longitude	latitude	housing_median_age	total_rooms
0	0	-122.23	37.88	41.0	880.0
1	1	-122.22	37.86	21.0	7099.0
2	2	-122.24	37.85	52.0	1467.0
3	3	-122.25	37.85	52.0	1274.0
4	4	-122.25	37.85	52.0	1627.0

```
test_set.head()
```

	index	longitude	latitude	housing_median_age	total_rooms
8	8	-122.26	37.84	42.0	2555.0
10	10	-122.26	37.85	52.0	2202.0
11	11	-122.26	37.85	52.0	3503.0
12	12	-122.26	37.85	52.0	2491.0
13	13	-122.26	37.84	52.0	696.0

```
# another option: scikit-learn splitters

from sklearn.model_selection import train_test_split

train_set, test_set = train_test_split(
    housing, test_size=0.2, random_state=42)
```

```
test_set.head()
```

	longitude	latitude	housing_median_age	total_rooms	tot
20046	-119.01	36.06	25.0	1505.0	Na
3024	-119.46	35.14	30.0	2943.0	Na
15663	-122.44	37.80	52.0	3830.0	Na
20484	-118.72	34.28	17.0	3051.0	Na
9814	-121.93	36.62	34.0	2351.0	Na

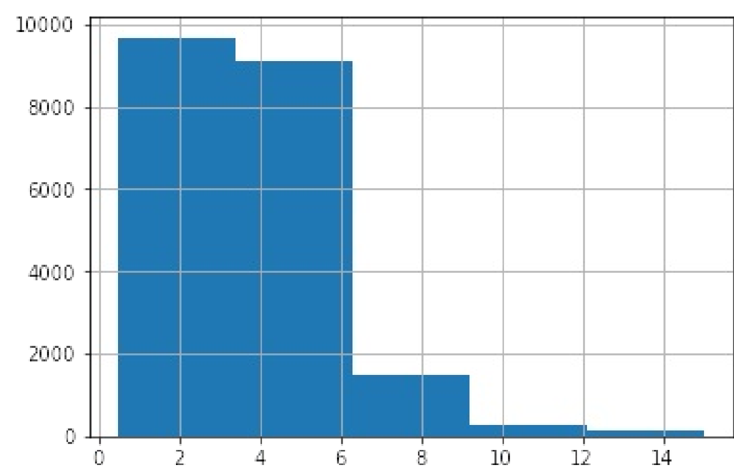
```
train_set.head()
```

	longitude	latitude	housing_median_age	total_rooms	tot
14196	-117.03	32.71	33.0	3126.0	627
8267	-118.16	33.77	49.0	3382.0	787
17445	-120.48	34.66	4.0	1897.0	331
14265	-117.11	32.69	36.0	1421.0	367
2271	-119.80	36.78	43.0	2382.0	431

```
# does sampling plan have a sampling bias?
# each strata in test dataset should mimic reality

housing['median_income'].hist(bins=5)
```

```
<matplotlib.axes._subplots.AxesSubplot at 0x7f15f7250588>
```




```
housing.describe()
```

	longitude	latitude	housing_median_age	total_r
count	20640.000000	20640.000000	20640.000000	20640.00
mean	-119.569704	35.631861	28.639486	2635.763
std	2.003532	2.135952	12.585558	2181.615
min	-124.350000	32.540000	1.000000	2.000000
25%	-121.800000	33.930000	18.000000	1447.750
50%	-118.490000	34.260000	29.000000	2127.000
75%	-118.010000	37.710000	37.000000	3148.000
max	-114.310000	41.950000	52.000000	39320.00

```
housing["income_cat"]=np.ceil(housing["median_income"]/1.5)

housing["income_cat"].where(housing["income_cat"]<5, 5.0, inplac
e=True)
```

```
housing.describe()
```

	longitude	latitude	housing_median_age	total_r
count	20640.000000	20640.000000	20640.000000	20640.00
mean	-119.569704	35.631861	28.639486	2635.763
std	2.003532	2.135952	12.585558	2181.615
min	-124.350000	32.540000	1.000000	2.000000
25%	-121.800000	33.930000	18.000000	1447.750
50%	-118.490000	34.260000	29.000000	2127.000
75%	-118.010000	37.710000	37.000000	3148.000
max	-114.310000	41.950000	52.000000	39320.00

```
from sklearn.model_selection import StratifiedShuffleSplit

split = StratifiedShuffleSplit(
    n_splits=1, test_size=0.2, random_state=42)

for train_index, test_index in split.split(housing, housing["inc
ome_cat"]):
    strat_train_set = housing.loc[train_index]
    strat_test_set  = housing.loc[test_index]
```

```
# review income category proportions
housing["income_cat"].value_counts() / len(housing)
```

```
3.0    0.350581
2.0    0.318847
4.0    0.176308
5.0    0.114438
1.0    0.039826
Name: income_cat, dtype: float64
```

```
# remove income_cat attribute (return dataset to original state)

for set in (strat_train_set, strat_test_set):
    set.drop(["income_cat"], axis=1, inplace=True)
```

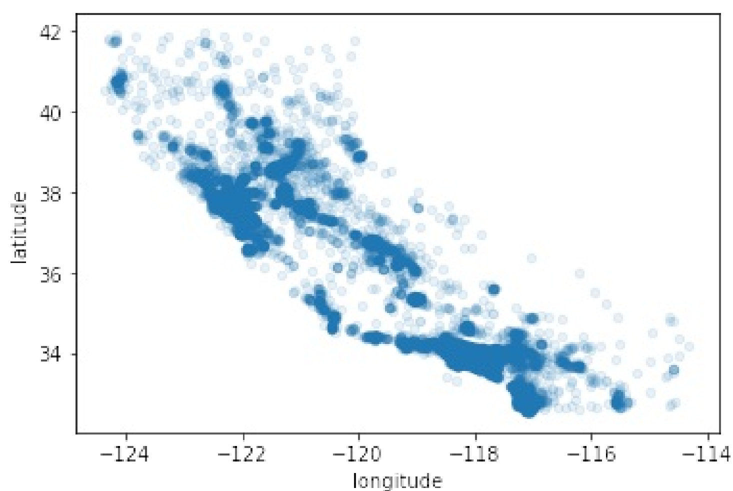
Visualization

```
housing = strat_train_set.copy()

# first: basic geographic distribution

housing.plot(kind="scatter", x="longitude", y="latitude", alpha=
0.1)
```

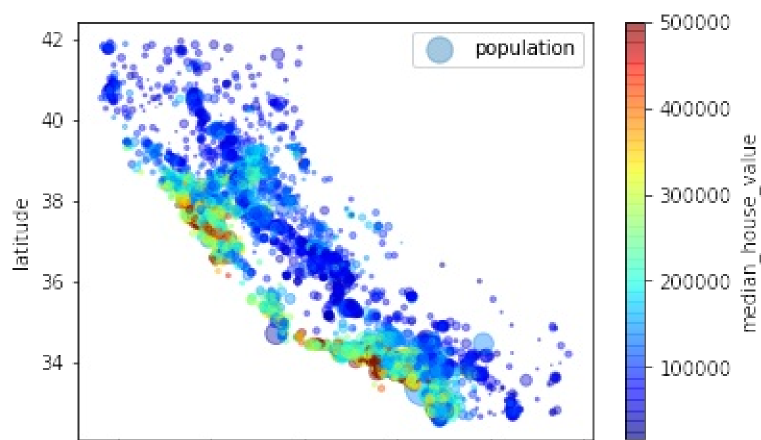
```
<matplotlib.axes._subplots.AxesSubplot at 0x7f15f71c4668>
```



```
# next: housing prices.
# color = price
# radius = population
# use predefined "jet" color map

housing.plot(
    kind="scatter",
    x="longitude",
    y="latitude",
    alpha=0.4,
    #s=housing["population"].apply(lambda n: n/100),
    s=housing["population"]/100,
    label="population",
    c="median_house_value",
    cmap=plt.get_cmap("jet"),
    colorbar=True,
)
plt.legend()
```

```
<matplotlib.legend.Legend at 0x7f15f5eff1d0>
```



Correlations

```
# next: look for correlatons to median house value.

corr_matrix = housing.corr()

corr_matrix['median_house_value'].sort_values(ascending=False)
```

```
median_house_value    1.000000
median_income         0.687160
total_rooms           0.135097
housing_median_age    0.114110
households            0.064506
total_bedrooms        0.047689
population           -0.026920
longitude            -0.047432
latitude             -0.142724
Name: median_house_value, dtype: float64
```

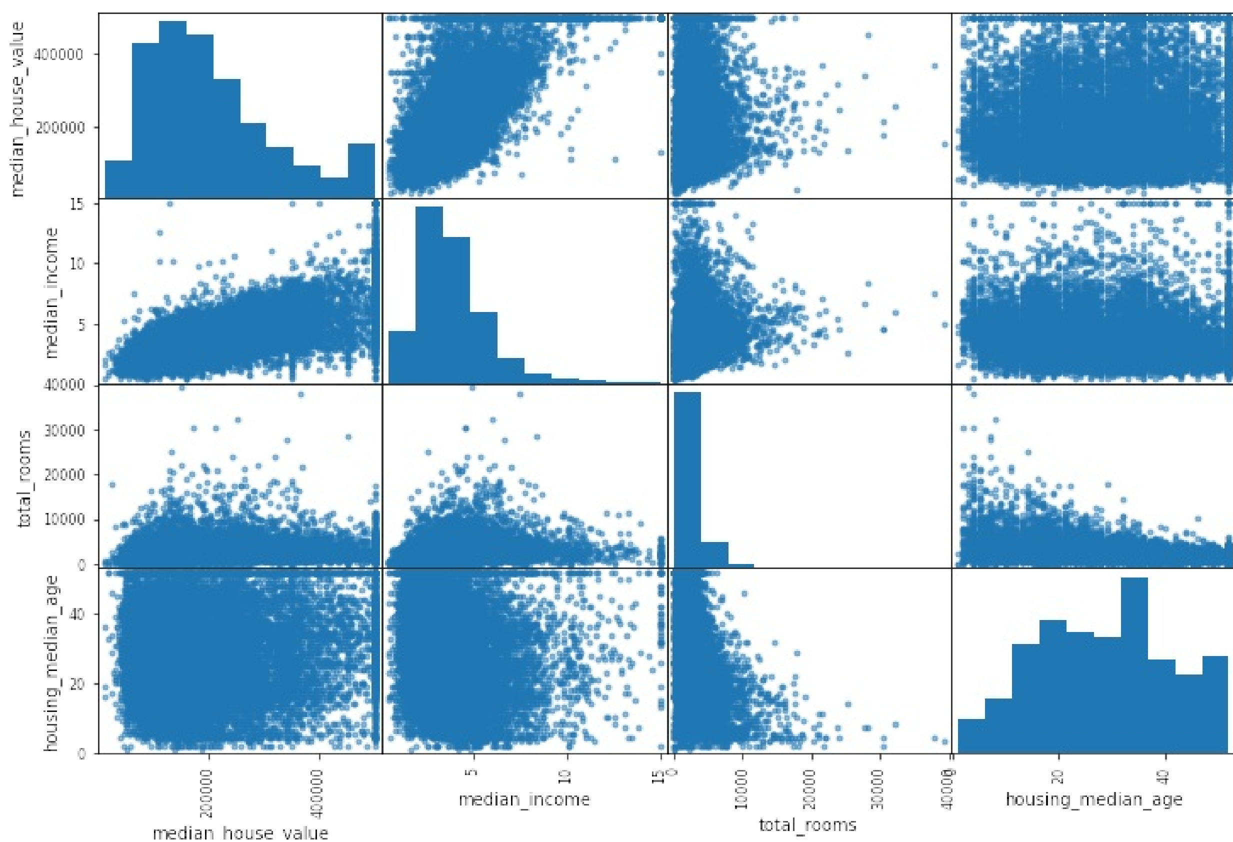
```
# another way of looking for correlations: scatter_matrix
# focus on top 3 factors from above

from pandas.tools.plotting import scatter_matrix

attributes = ["median_house_value", "median_income", "total_room
s",
"housing_median_age"]

scatter_matrix(housing[attributes], figsize=(12, 8))
```

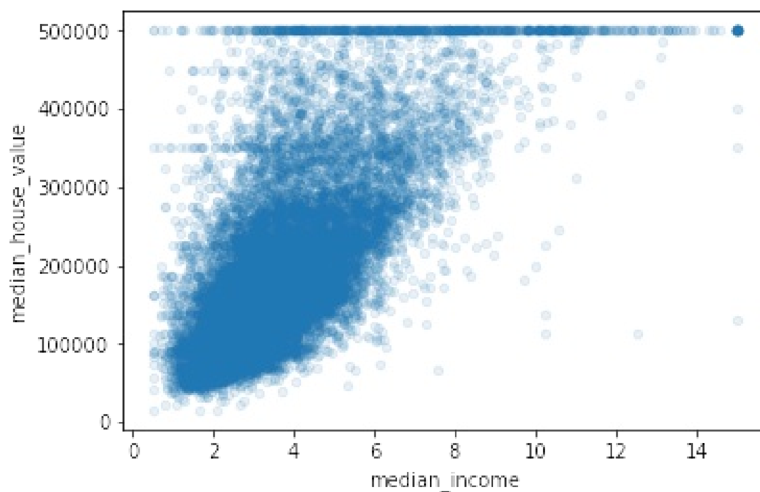
```
array([[<matplotlib.axes._subplots.AxesSubplot object at 0x7f15f
5fc20f0>,
        <matplotlib.axes._subplots.AxesSubplot object at 0x7f15f
5eda860>,
        <matplotlib.axes._subplots.AxesSubplot object at 0x7f15f
602f898>,
        <matplotlib.axes._subplots.AxesSubplot object at 0x7f15f
5ff2860>],
       [<matplotlib.axes._subplots.AxesSubplot object at 0x7f15f
5f7dd68>,
        <matplotlib.axes._subplots.AxesSubplot object at 0x7f15f
5e836d8>,
        <matplotlib.axes._subplots.AxesSubplot object at 0x7f15f
460e710>,
        <matplotlib.axes._subplots.AxesSubplot object at 0x7f15f
45d50b8>],
       [<matplotlib.axes._subplots.AxesSubplot object at 0x7f15f
45224e0>,
        <matplotlib.axes._subplots.AxesSubplot object at 0x7f15f
454ab38>,
        <matplotlib.axes._subplots.AxesSubplot object at 0x7f15f
449ec88>,
        <matplotlib.axes._subplots.AxesSubplot object at 0x7f15f
44e5898>],
       [<matplotlib.axes._subplots.AxesSubplot object at 0x7f15f
44380b8>,
        <matplotlib.axes._subplots.AxesSubplot object at 0x7f15f
4450be0>,
        <matplotlib.axes._subplots.AxesSubplot object at 0x7f15f
43c2da0>,
        <matplotlib.axes._subplots.AxesSubplot object at 0x7f15f
43960f0>]], dtype=object)
```




```
# median house value to median income seems to be the most promi
sing.
# let's zoom in.

housing.plot(
    kind="scatter", x="median_income", y="median_house_value",
    alpha=0.1)
```

```
<matplotlib.axes._subplots.AxesSubplot at 0x7f15f41b1a90>
```




```
# combine some attributes to create more useful ones
# then rebuild the correlation matrix.

housing["rooms_per_household"] = housing["total_rooms"]/housing[
"households"]
housing["bedrooms_per_room"] = housing["total_bedrooms"]/housing[
"total_rooms"]
housing["population_per_household"]=housing["population"]/housin
g["households"]

corr_matrix = housing.corr()
corr_matrix['median_house_value'].sort_values(ascending=False)

# *** NOTE: rooms_per_household corr (in book) show more improve
ment, ~0.199
# compared to our 0.146. Not sure of root cause yet. ***
```

```
median_house_value          1.000000
median_income               0.687160
rooms_per_household         0.146285
total_rooms                 0.135097
housing_median_age          0.114110
households                  0.064506
total_bedrooms              0.047689
population_per_household   -0.021985
population                 -0.026920
longitude                  -0.047432
latitude                   -0.142724
bedrooms_per_room          -0.259984
Name: median_house_value, dtype: float64
```

Data Cleanup

```
# revert to clean copy of stratified training dataset
# separate predictors from labels

housing = strat_train_set.drop("median_house_value", axis=1)

housing_labels = strat_train_set["median_house_value"].copy()
```

```
# 'total bedrooms' has some missing values - fix
# can use DataFrame dropna(), drop(), fillna()

# use Scikit-Learn class to handle missing values
from sklearn.preprocessing import Imputer
imputer = Imputer(strategy="median")
```

```
# drop ocean_proximity attribute, since it's non-numeric.
# then fit to training data.

housing_num = housing.drop("ocean_proximity", axis=1)
imputer.fit(housing_num)
```

```
Imputer(axis=0, copy=True, missing_values='NaN', strategy='media
n', verbose=0)
```

```
# now what do we have?
imputer.statistics_
```

```
array([ -118.51  ,    34.26  ,    29.    ,  2119.5   ,   433.
 ,
        1164.    ,   408.    ,     3.5409])
```

```
housing_num.median().values
```

```
array([ -118.51  ,    34.26  ,    29.    ,  2119.5   ,   433.
 ,
        1164.    ,   408.    ,     3.5409])
```

```
# update training set by replacing missing values with learned m
edians
X = imputer.transform(housing_num)
```

```
pd.DataFrame(X, columns=housing_num.columns).info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 16512 entries, 0 to 16511
Data columns (total 8 columns):
longitude             16512 non-null float64
latitude              16512 non-null float64
housing_median_age    16512 non-null float64
total_rooms           16512 non-null float64
total_bedrooms        16512 non-null float64
population            16512 non-null float64
households            16512 non-null float64
median_income         16512 non-null float64
dtypes: float64(8)
memory usage: 1.0 MB
```

```
# convert ocean_proximity feature to numbers using LabelEncoder.

from sklearn.preprocessing import LabelEncoder
encoder = LabelEncoder()

housing_cat = housing['ocean_proximity']
housing_cat_encoded = encoder.fit_transform(housing_cat)
housing_cat_encoded
```

```
array([0, 0, 4, ..., 1, 0, 3])
```

```
# how is 'ocean_proximity' mapped?
print(encoder.classes_)
```

```
['<1H OCEAN' 'INLAND' 'ISLAND' 'NEAR BAY' 'NEAR OCEAN']
```

```
# a better solution for categorical data: one-hot encoding

from sklearn.preprocessing import OneHotEncoder
encoder = OneHotEncoder()

# output = SciPy sparse matrix, better for memory usage
# if you need a dense NumPy array, call toarray()

housing_cat_1hot = encoder.fit_transform(housing_cat_encoded.res
hape(-1,1))
housing_cat_1hot
```

```
<16512x5 sparse matrix of type '<class 'numpy.float64'>'
    with 16512 stored elements in Compressed Sparse Row format>
```

Label Binarization:

- A shortcut (text categories => integer categories => one-hot vectors)

```
from sklearn.preprocessing import LabelBinarizer
encoder = LabelBinarizer()

housing_cat_1hot = encoder.fit_transform(housing_cat)
housing_cat_1hot
```

```
array([[1, 0, 0, 0, 0],
       [1, 0, 0, 0, 0],
       [0, 0, 0, 0, 1],
       ...,
       [0, 1, 0, 0, 0],
       [1, 0, 0, 0, 0],
       [0, 0, 0, 1, 0]])
```

Custom Transformers:

- Create your own using SciKit-Learn classes
- implement fit(), transform() and fit_transform() methods
- (fit_transform comes for free by using TransformerMixin as a base class.)

```
from sklearn.base import BaseEstimator, TransformerMixin

rooms_ix, bedrooms_ix, population_ix, household_ix = 3, 4, 5, 6

class CombinedAttributesAdder(BaseEstimator, TransformerMixin):

    def __init__(self, add_bedrooms_per_room = True): # no *args or **kargs

        self.add_bedrooms_per_room = add_bedrooms_per_room

    def fit(self, X, y=None):
        return self # nothing else to do

    def transform(self, X, y=None):
        rooms_per_household      = X[:, rooms_ix] / X[:, household_ix]
        population_per_household = X[:, population_ix] / X[:, household_ix]

        if self.add_bedrooms_per_room:
            bedrooms_per_room = X[:, bedrooms_ix] / X[:, rooms_ix]
            return np.c_[X,
                         rooms_per_household,
                         population_per_household,
                         bedrooms_per_room]
        else:
            return np.c_[X,
                         rooms_per_household,
                         population_per_household]

attr_adder = CombinedAttributesAdder(add_bedrooms_per_room=False)

housing_extra_attribs = attr_adder.transform(housing.values)
```

Feature Scaling

- Min-max scaling (normalization) = shift & rescale to [0,1]
- SciKit MinMaxScaler will do this for you.
- Standardization subtracts mean & divides by variance - result has unit variance
- SciKit StandardScaler does this for you.

Pipelining

- SciKit Pipeline class helps to standardize the sequence of transforms you need for your project.
- Pipelines = list of estimator steps. All but the last must be transformers (they must have fit_transform() method.)

```
# "DataFrameSelector" is a custom transformer class.
# grabs the specified feature, drops the rest, converts the DF i
nto a NumPy array.

from sklearn.base import BaseEstimator, TransformerMixin

class DataFrameSelector(BaseEstimator, TransformerMixin):
    def __init__ (self, attribute_names):
        self.attribute_names = attribute_names

    def fit (self, X, y=None):
        return self

    def transform (self, X):
        return X[self.attribute_names].values
```

```
from sklearn.pipeline import Pipeline, FeatureUnion
from sklearn.preprocessing import StandardScaler

num_attribs = list(housing_num)
cat_attribs = ['ocean_proximity']

num_pipeline = Pipeline([
    ('selector',      DataFrameSelector(num_attribs)),
    ('imputer',       Imputer(strategy="median")),
    ('attribs_adder', CombinedAttributesAdder()),
    ('std_scaler',    StandardScaler()),
    ])

cat_pipeline = Pipeline([
    ('selector',      DataFrameSelector(cat_attribs)),
    ('label_binarizer', LabelBinarizer()),
])

full_pipeline = FeatureUnion(transformer_list =[
    ('num_pipeline', num_pipeline),
    ('cat_pipeline', cat_pipeline)
])
```

```
# let's try it out:

housing_prepared = full_pipeline.fit_transform(housing)
housing_prepared
```

```
array([[-1.15604281,  0.77194962,  0.74333089, ...,  0.        ,
         0.        ,  0.        ],
       [-1.17602483,  0.6596948 , -1.1653172 , ...,  0.        ,
         0.        ,  0.        ],
       [ 1.18684903, -1.34218285,  0.18664186, ...,  0.        ,
         0.        ,  1.        ],
       ...,
       [ 1.58648943, -0.72478134, -1.56295222, ...,  0.        ,
         0.        ,  0.        ],
       [ 0.78221312, -0.85106801,  0.18664186, ...,  0.        ,
         0.        ,  0.        ],
       [-1.43579109,  0.99645926,  1.85670895, ...,  0.        ,
         1.        ,  0.        ]])
```

```
housing_prepared.shape
```

```
(16512, 16)
```

- Note: pip3 install sklearn-pandas => gets a DataFrameMapper class

Model Selection & Training

```
# let's start with a linear regression

from sklearn.linear_model import LinearRegression

lin_reg = LinearRegression()
lin_reg.fit(housing_prepared, housing_labels)
```

```
LinearRegression(copy_X=True, fit_intercept=True, n_jobs=1, norm
alize=False)
```

```
# first try. NOT very accurate.

some_data          = housing.iloc[:5]
some_labels        = housing_labels.iloc[:5]
some_data_prepared = full_pipeline.transform(some_data)

print ("predictions:\t", lin_reg.predict(some_data_prepared))
print ("labels:\t", list(some_labels))
```

```
predictions:     [ 210644.60459286  317768.80697211  210956.4333
1178   59218.98886849
  189747.55849879]
labels:     [286600.0, 340600.0, 196900.0, 46300.0, 254500.0]
```

```
# why? look at RMSE on whole training set.

from sklearn.metrics import mean_squared_error

housing_predictions = lin_reg.predict(housing_prepared)
lin_mse             = mean_squared_error(housing_labels, housing
_predictions)
lin_rmse            = np.sqrt(lin_mse)

print ("typical prediction error:\t", lin_rmse)
```

```
typical prediction error:     68628.1981985
```

```
# Hmmm. Not good. Underfit situation.
# Let's try a more powerful model, like a Decision Tree.

from sklearn.tree import DecisionTreeRegressor

tree_reg = DecisionTreeRegressor()
tree_reg.fit(housing_prepared, housing_labels)
```

```
DecisionTreeRegressor(criterion='mse', max_depth=None, max_featu
res=None,
           max_leaf_nodes=None, min_impurity_split=1e-07,
           min_samples_leaf=1, min_samples_split=2,
           min_weight_fraction_leaf=0.0, presort=False, random_s
tate=None,
           splitter='best')
```

```
# Zero error? No way...

housing_predictions = tree_reg.predict(housing_prepared)
tree_mse = mean_squared_error(housing_labels, housing_prediction
s)
tree_rmse = np.sqrt(tree_mse)

print ("typical prediction error:\t", tree_rmse)
```

```
typical prediction error:     0.0
```

```
# Use K-fold cross-validation
# Train & eval Decision Tree model against 10 splits of training
 dataset
# Returns 10 evaluation scores.

from sklearn.model_selection import cross_val_score

scores = cross_val_score(
    tree_reg,
    housing_prepared,
    housing_labels,
    scoring="neg_mean_squared_error",
    cv=10)

rmse_scores = np.sqrt(-scores)

def display_scores(scores):
    print("Scores:", scores)
    print("Mean:", scores.mean())
    print("Standard deviation:", scores.std())

display_scores(rmse_scores)
```

```
Scores: [ 69368.62190153  66248.56520386  72284.6557095   68417.
57732406
  70049.44916939  74941.75765797  70236.59348749  69466.63688954
  76140.22952307  70217.59755116]
Mean: 70737.1684418
Standard deviation: 2815.58298405
```

```
# So, Decision Tree RMSE: mean ~71097, stdev 2165 (still sucks.)
# compare to earlier Linear Regression:

lin_scores = cross_val_score(
    lin_reg,
    housing_prepared,
    housing_labels,
    scoring="neg_mean_squared_error",
    cv=10)

lin_rmse_scores = np.sqrt(-lin_scores)

display_scores(lin_rmse_scores)
```

```
Scores: [ 66782.73843989  66960.118071    70347.95244419  74739.
57052552
  68031.13388938  71193.84183426  64969.63056405  68281.61137997
  71552.91566558  67665.10082067]
Mean: 69052.4613635
Standard deviation: 2731.6740018
```

```
# Yep, DT overfit is just about as bad. (RMSE mean 69052, stdev
2731)
# Let's try a RandomForest.

from sklearn.ensemble import RandomForestRegressor

forest_reg = RandomForestRegressor()
forest_reg.fit(housing_prepared, housing_labels)

forest_scores = cross_val_score(
    forest_reg,
    housing_prepared,
    housing_labels,
    scoring="neg_mean_squared_error",
    cv=10)

forest_rmse_scores = np.sqrt(-forest_scores)

display_scores(forest_rmse_scores)
```

```
Scores: [ 52480.82629458  50035.41358467  53747.69332484  55053.
95194112
  51800.65152945  55919.01705209  52226.75176017  50912.82366116
  55708.47271341  51931.81080304]
Mean: 52981.7412665
Standard deviation: 1929.32402243
```

```
# OK, RandomForest is a little better.
# RMSE mean ~52495, stdev ~1569
```

Fine-Tuning Model with Grid Search of Hyperparameters

```
from sklearn.model_selection import GridSearchCV

param_grid = [
    {'n_estimators': [3, 10, 30],
     'max_features': [2, 4, 6, 8]},
    {'bootstrap': [False], # bootstrap = True = default setting
     'n_estimators': [3, 10],
     'max_features': [2, 3, 4]},
]

forest_reg  = RandomForestRegressor()

grid_search = GridSearchCV(
    forest_reg,
    param_grid,
    cv=5,
    scoring = 'neg_mean_squared_error')

grid_search.fit(housing_prepared, housing_labels)
```

```
GridSearchCV(cv=5, error_score='raise',
       estimator=RandomForestRegressor(bootstrap=True, criterion
='mse', max_depth=None,
           max_features='auto', max_leaf_nodes=None,
           min_impurity_split=1e-07, min_samples_leaf=1,
           min_samples_split=2, min_weight_fraction_leaf=0.0,
           n_estimators=10, n_jobs=1, oob_score=False, random_st
ate=None,
           verbose=0, warm_start=False),
       fit_params={}, iid=True, n_jobs=1,
       param_grid=[{'max_features': [2, 4, 6, 8], 'n_estimators'
: [3, 10, 30]}, {'bootstrap': [False], 'max_features': [2, 3, 4]
, 'n_estimators': [3, 10]}],
       pre_dispatch='2*n_jobs', refit=True, return_train_score=T
rue,
       scoring='neg_mean_squared_error', verbose=0)
```

```
# Best combination of parameters?
grid_search.best_params_
```

```
{'max_features': 6, 'n_estimators': 30}
```

```
# Best estimator?
grid_search.best_estimator_
```

```
RandomForestRegressor(bootstrap=True, criterion='mse', max_depth
=None,
           max_features=6, max_leaf_nodes=None, min_impurity_spl
it=1e-07,
           min_samples_leaf=1, min_samples_split=2,
           min_weight_fraction_leaf=0.0, n_estimators=30, n_jobs
=1,
           oob_score=False, random_state=None, verbose=0, warm_s
tart=False)
```

```
# Evaluation scores:

cvres = grid_search.cv_results_

for mean_score, params in zip(cvres["mean_test_score"],
                              cvres["params"]):
    print(np.sqrt(-mean_score), params)
```

```
63492.9975584 {'max_features': 2, 'n_estimators': 3}
55677.1037862 {'max_features': 2, 'n_estimators': 10}
52917.801725 {'max_features': 2, 'n_estimators': 30}
60442.2787178 {'max_features': 4, 'n_estimators': 3}
53209.7111283 {'max_features': 4, 'n_estimators': 10}
50621.1191846 {'max_features': 4, 'n_estimators': 30}
58591.8196313 {'max_features': 6, 'n_estimators': 3}
52353.3606044 {'max_features': 6, 'n_estimators': 10}
49838.3807 {'max_features': 6, 'n_estimators': 30}
58615.6100561 {'max_features': 8, 'n_estimators': 3}
51726.2593734 {'max_features': 8, 'n_estimators': 10}
50074.3050139 {'max_features': 8, 'n_estimators': 30}
62010.5215854 {'bootstrap': False, 'max_features': 2, 'n_estimat
ors': 3}
54852.7770725 {'bootstrap': False, 'max_features': 2, 'n_estimat
ors': 10}
60246.2164711 {'bootstrap': False, 'max_features': 3, 'n_estimat
ors': 3}
52752.4109521 {'bootstrap': False, 'max_features': 3, 'n_estimat
ors': 10}
58355.1846204 {'bootstrap': False, 'max_features': 4, 'n_estimat
ors': 3}
51724.6800894 {'bootstrap': False, 'max_features': 4, 'n_estimat
ors': 10}
```

```
# best solution:
# max_features = 6, n_estimators = 30 (RMSE ~49,960)
```

```
feature_importances = grid_search.best_estimator_.feature_import
ances_
feature_importances
```

```
array([  8.00229340e-02,   7.13499357e-02,   4.21346911e-02,
         1.73340009e-02,   1.55694906e-02,   1.76527489e-02,
         1.56813711e-02,   3.21068169e-01,   7.54675530e-02,
         1.07645094e-01,   5.74608930e-02,   1.47327045e-02,
         1.57310792e-01,   9.20951468e-05,   2.63317542e-03,
         3.84435167e-03])
```

```
# display feature "importance" scores next to their names:

extra_attribs       = ["rooms_per_hhold", "pop_per_hhold", "bedr
ooms_per_room"]
cat_one_hot_attribs = list(encoder.classes_)
attributes          = num_attribs + extra_attribs + cat_one_hot_
attribs

sorted(zip(feature_importances, attributes), reverse=True)
```

```
[(0.32106816893273865, 'median_income'),
 (0.15731079177984286, 'INLAND'),
 (0.10764509417315272, 'pop_per_hhold'),
 (0.080022934000105003, 'longitude'),
 (0.075467553036607335, 'rooms_per_hhold'),
 (0.071349935674308126, 'latitude'),
 (0.057460893036370447, 'bedrooms_per_room'),
 (0.04213469106714228, 'housing_median_age'),
 (0.017652748894983483, 'population'),
 (0.017334000890698829, 'total_rooms'),
 (0.015681371107232313, 'households'),
 (0.015569490624941605, 'total_bedrooms'),
 (0.014732704544371122, '<1H OCEAN'),
 (0.0038443516681782959, 'NEAR OCEAN'),
 (0.00263317542255579, 'NEAR BAY'),
 (9.2095146771177451e-05, 'ISLAND')]
```

Time to Eval System on Test dataset

```
final_model = grid_search.best_estimator_

X_test          = strat_test_set.drop("median_house_value", axis=
1)
y_test          = strat_test_set["median_house_value"].copy()
X_test_prepared = full_pipeline.transform(X_test)

final_predictions = final_model.predict(X_test_prepared)
final_mse = mean_squared_error(y_test, final_predictions)
final_rmse = np.sqrt(final_mse)
final_rmse
```

```
47574.62166586089
```

MNIST: the "Hello World" of Machine Learning

```
# Alternative local file loader (due to mldata.org being down)

from scipy.io import loadmat
mnist_raw = loadmat("mnist-original.mat")
mnist = {
    "data": mnist_raw["data"].T,
    "target": mnist_raw["label"][0],
    "COL_NAMES": ["label", "data"],
    "DESCR": "mldata.org dataset: mnist-original",
    }
```

```
# 70K images, 28x28 pixels/image, each pixel = 0 (white) to 255
(black)
mnist # a dict object
```

```
{'COL_NAMES': ['label', 'data'],
 'DESCR': 'mldata.org dataset: mnist-original',
 'data': array([[0, 0, 0, ..., 0, 0, 0],
        [0, 0, 0, ..., 0, 0, 0],
        [0, 0, 0, ..., 0, 0, 0],
        ...,
        [0, 0, 0, ..., 0, 0, 0],
        [0, 0, 0, ..., 0, 0, 0],
        [0, 0, 0, ..., 0, 0, 0]], dtype=uint8),
 'target': array([ 0.,  0.,  0., ...,  9.,  9.,  9.])}
```

```
# take a peek
X,y = mnist['data'], mnist['target']

X.shape, y.shape
```

```
((70000, 784), (70000,))
```

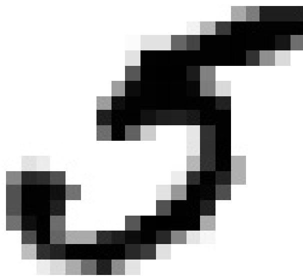
```
# display example image

%matplotlib inline
import matplotlib
import matplotlib.pyplot as plt

some_digit = X[36000]
some_digit_image = some_digit.reshape(28, 28)

plt.imshow(
    some_digit_image,
    cmap = matplotlib.cm.binary,
    interpolation="nearest")

plt.axis("off")
plt.show()
```



```
# looks like a "five". What's the corresponding label?
y[36000]
```

```
5.0
```

```
# dataset already split into training (1st 60K) & test (last 10K
) images.
# shuffle training set for cross-validation quality

X_train, X_test, y_train, y_test = X[:60000], X[60000:], y[:60000
], y[60000:]

import numpy as np

shuffle_index = np.random.permutation(60000)
X_train, y_train = X_train[shuffle_index], y_train[shuffle_index
]

print(X_train.shape, X_test.shape, y_train.shape, y_test.shape)
```

```
(60000, 784) (10000, 784) (60000,) (10000,)
```

Binary classifier training - distinguish between 2 classes

- Using Stochastic Descent

```
# Start by only trying to ID "five" digits.

y_train_5 = (y_train == 5) # create target vectors
y_test_5  = (y_test == 5)

print(y_train_5.shape, y_train_5)
print(y_test_5.shape, y_test_5)

# SGD classifier: good at handling large DBs
#                 also good at handling one-at-a-time learning

from sklearn.linear_model import SGDClassifier
sgd_clf = SGDClassifier(random_state=42)
sgd_clf.fit(X_train, y_train_5)

# did it correctly predict the "five" found above?
print(sgd_clf.predict([some_digit]))
```

```
(60000,) [False False False ..., False False False]
(10000,) [False False False ..., False False False]
[ True]
```

Performance Measures

```
# measure accuracy using K-fold (n=3) cross-validation scores

from sklearn.model_selection import cross_val_score

print(cross_val_score(
        sgd_clf,
        X_train,
        y_train_5,
        cv=3,
        scoring="accuracy"))

# 90% accuracy = pretty easy when 90% of digits aren't fives to
begin with ... :-|
```

```
[ 0.96795  0.96975  0.96855]
```

```
# rolling your own cross-validation. Results should be similar-i
sh to above.

from sklearn.model_selection import StratifiedKFold
from sklearn.base import clone

skfolds = StratifiedKFold(n_splits=3, random_state=42)

for train_index, test_index in skfolds.split(X_train, y_train_5)
:

    clone_clf = clone(sgd_clf)

    X_train_folds =  X_train[train_index]
    y_train_folds = (y_train_5[train_index])
    X_test_fold =    X_train[test_index]
    y_test_fold =   (y_train_5[test_index])

    clone_clf.fit(X_train_folds, y_train_folds)

    y_pred = clone_clf.predict(X_test_fold)

    n_correct = sum(y_pred == y_test_fold)
    print(n_correct / len(y_pred))
```

```
0.96795
0.96975
0.96855
```

```
# 95% accuracy sounds too good to be true. How about not-fives?

from sklearn.base import BaseEstimator

class Never5Classifier(BaseEstimator):
    def fit(self, X, y=None):
        pass

    def predict(self, X):
        return np.zeros((len(X), 1), dtype=bool)

never_5_clf = Never5Classifier()

print(cross_val_score(
    never_5_clf,
    X_train,
    y_train_5,
    cv=3,
    scoring="accuracy"))
```

```
[ 0.9096   0.9124   0.90695]
```

```
# only ~10% of images are "five", so ~90% of images are "not fiv
e".
# You SHOULD be right about 90% of the time. :-)

# Lesson Learned:
# Accuracy not a good metric for classifiers - esp those with sk
ewed datasets.
```

Confusion Matrix - a better way of evaluating a classifier

```
# general idea: count #times instances of A are classified as B.
# first, need a set of predictions.

from sklearn.model_selection import cross_val_predict

# Generate cross-val'd predictions for each datapoint
y_train_pred = cross_val_predict(sgd_clf, X_train, y_train_5, cv=
3)

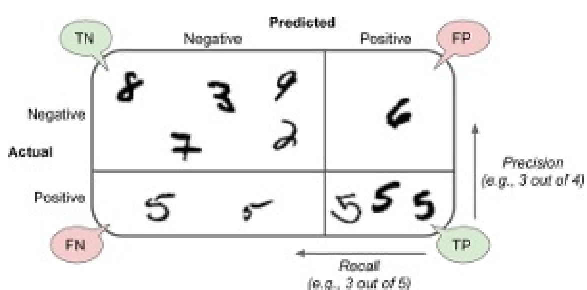
# ROWS = actual classes
# COLS = predicted classes

from sklearn.metrics import confusion_matrix

print(confusion_matrix(y_train_5, y_train_pred))
```

```
[[54044   535]
 [ 1340  4081]]
```

Classifier metrics: precision = TP/(TP+FP); recall (sensitivity) = TP/(TP+FN)




```
print(3841 / (3841+1515), 3841/(3841+1580))
```

```
0.7171396564600448 0.7085408596199964
```

```
# precision, recall, f1 metrics
# precision/recall tradeoff: increasing one reduces the other.

from sklearn.metrics import precision_score, recall_score, f1_sc
ore
print("precision:\n",precision_score(y_train_5, y_train_pred))
print("recall:\n",recall_score(y_train_5, y_train_pred))

# F1 score favors classifiers with similar precision & recall.
print("f1:\n",f1_score(y_train_5, y_train_pred))
```

```
precision:
 0.884098786828
recall:
 0.752813134108
f1:
 0.813191192587
```

Precision/Recall Tradeoffs

```
# Scikit doesn't let you directly set threshold values (which dr
ive the decision
# function for precision/recall.) But you can use the decision f
unction itself.

y_scores = sgd_clf.decision_function([some_digit])
print(y_scores)

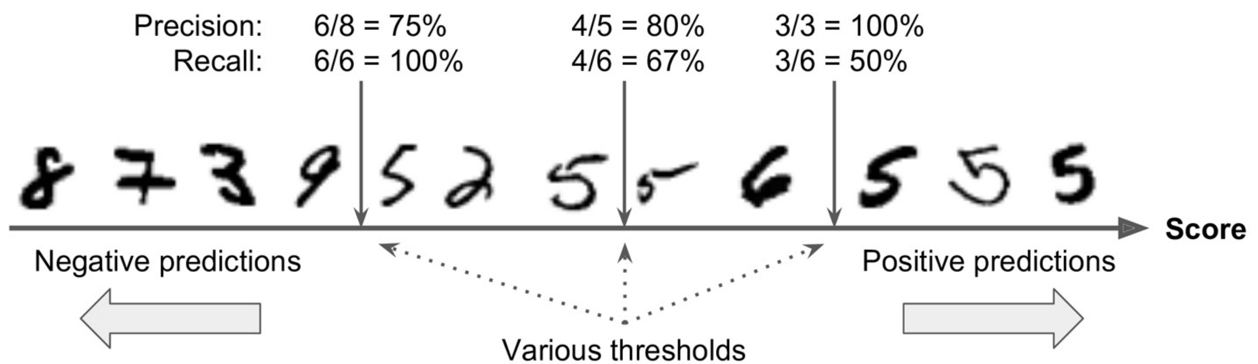
threshold = 0
y_some_digit_pred = (y_scores > threshold)
print(y_some_digit_pred)
```

```
[ 57844.42736708]
[ True]
```

```
# raising the threshold reduces recall...

threshold = 200000
y_some_digit_pred = (y_scores > threshold)
print(y_some_digit_pred)
```

```
[False]
```



```
# how to find the right threshold?
# start with getting decision scores instead of predictions.

y_scores = cross_val_predict(
    sgd_clf,
    X_train,
    y_train_5,
    cv=3,
    method="decision_function")

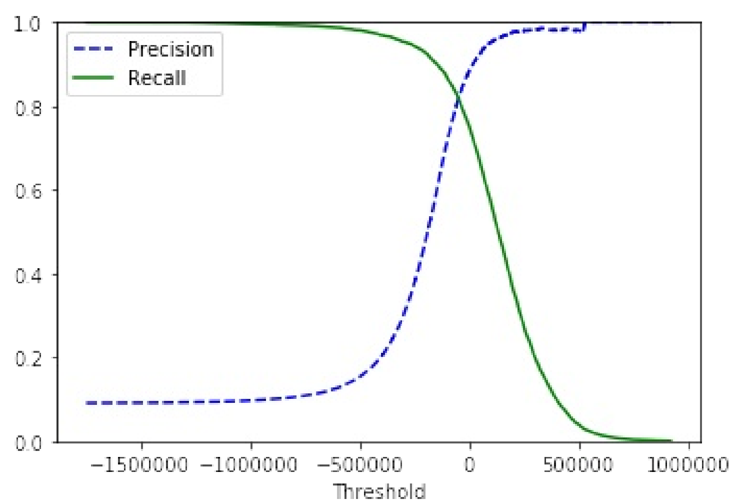
# use results to build a precision/recall curve

from sklearn.metrics import precision_recall_curve
precisions, recalls, thresholds = precision_recall_curve(y_train
_5, y_scores)

# plot the result

def plot_precision_recall_vs_threshold(precisions, recalls, thre
sholds):
    plt.plot(thresholds,
             precisions[:-1],
             "b--",
             label="Precision")
    plt.plot(thresholds,
             recalls[:-1],
             "g-",
             label="Recall")
    plt.xlabel("Threshold")
    plt.legend(loc="upper left")
    plt.ylim([0, 1])

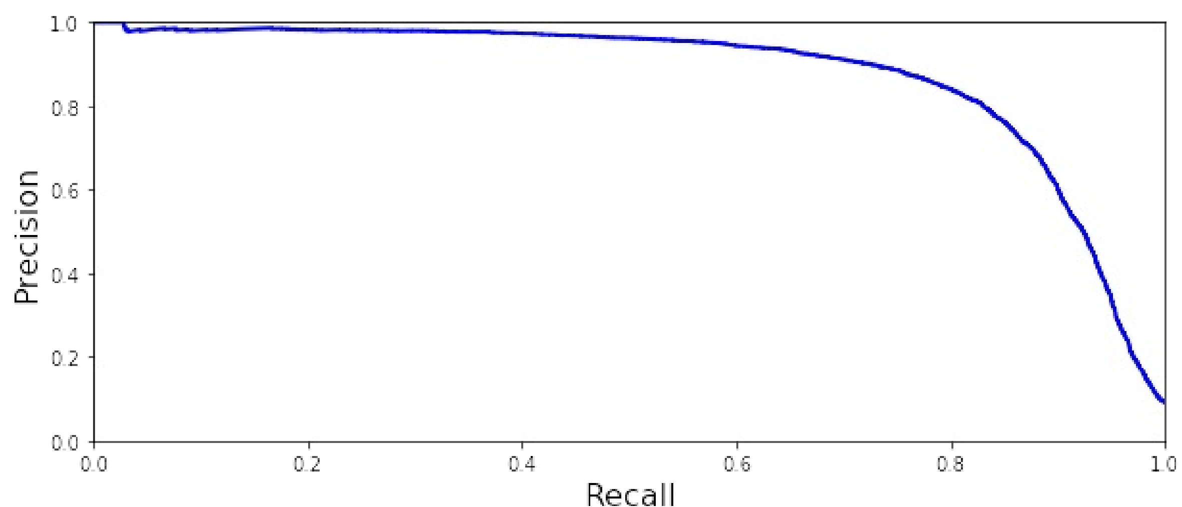
plot_precision_recall_vs_threshold(precisions, recalls, threshol
ds)
plt.show()
```




```
# plot precision vs recall to look for knee of the curve

def plot_precision_vs_recall(precisions, recalls):
    plt.plot(recalls, precisions, "b-", linewidth=2)
    plt.xlabel("Recall", fontsize=16)
    plt.ylabel("Precision", fontsize=16)
    plt.axis([0, 1, 0, 1])

plt.figure(figsize=(10, 4))
plot_precision_vs_recall(precisions, recalls)
plt.show()
```



```
# assume you're targeting 90% precision:
# guesswork from precision-recall curve suggests setting thresho
ld ~50000

y_train_pred_90 = (y_scores > 50000)
print(y_train_pred_90.shape, y_train_pred_90)

print("precision:\n",precision_score(y_train_5, y_train_pred_90)
)
print("recall:\n",recall_score(y_train_5, y_train_pred_90))
```

```
(60000,) [False False False ..., False False False]
precision:
 0.924948770492
recall:
 0.666113263236
```

ROC (Receiver Operating Characteristic) curve

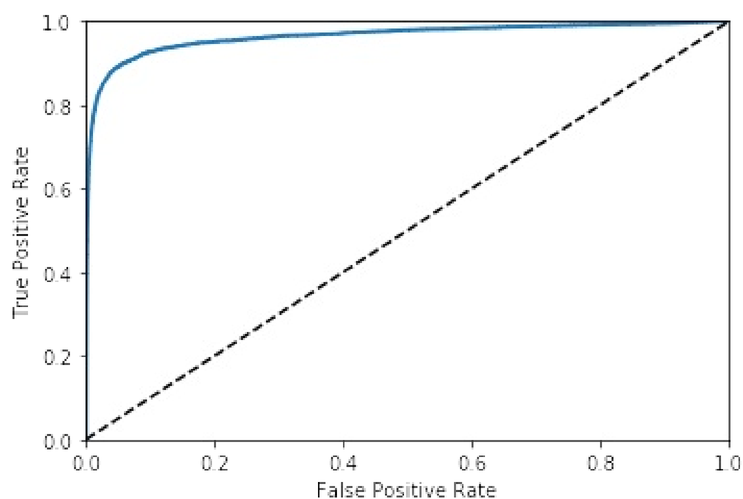
```
# ROC plots TRUE POSITIVE rate (TP = recall) vs FALSE POSITIVE r
ate. (FP = 1-specificity)

from sklearn.metrics import roc_curve
fpr, tpr, thresholds = roc_curve(y_train_5, y_scores)

def plot_roc_curve(fpr, tpr, label=None):
    plt.plot(fpr, tpr, linewidth=2, label=label)
    plt.plot([0, 1], [0, 1], 'k--')
    plt.axis([0, 1, 0, 1])
    plt.xlabel('False Positive Rate')
    plt.ylabel('True Positive Rate')

plot_roc_curve(fpr, tpr)
plt.show()

# tradeoff: higher recall (TP) => more false positives produced.
# dotted line = purely random classifier results.
```




```
# area under curve (AUC) metric:
# perfect score = ROC AUC = 1.0
# random score = ROC AUC = 0.5

from sklearn.metrics import roc_auc_score
print(roc_auc_score(y_train_5, y_scores))
```

```
0.964880839199
```

```
# train Random Forest classifier
# compare its ROC curve & AUC to SGD classifier

from sklearn.ensemble import RandomForestClassifier

forest_clf = RandomForestClassifier(random_state=42)

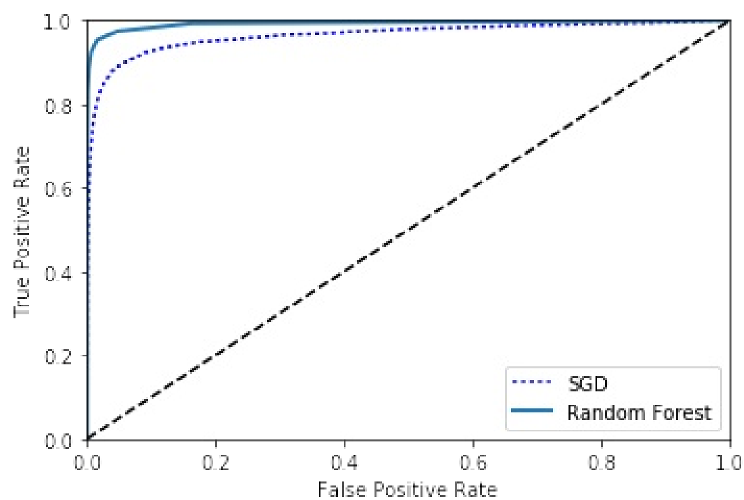
# Random Forest doesn't have decision_function(); use predict_pr
oba() instead.
# returns array (row per instance, column per class)

y_probas_forest = cross_val_predict(
    forest_clf,
    X_train,
    y_train_5,
    cv=3,
    method="predict_proba")

# To plot ROC curve, you need scores - not probabilities.
# use positive class probability as the score.

y_scores_forest = y_probas_forest[:, 1]
fpr_forest, tpr_forest, thresholds_forest = roc_curve(y_train_5,
y_scores_forest)

# plot ROC curve
plt.plot(fpr, tpr, "b:", label="SGD")
plot_roc_curve(fpr_forest, tpr_forest, "Random Forest")
plt.legend(loc="lower right")
plt.show()
```




```
# Random Forest curve looks much steeper (better). How's the ROC
 AUC score?
print(roc_auc_score(y_train_5, y_scores_forest))

# How's the precision & recall?
y_train_pred_forest = cross_val_predict(
    forest_clf,
    X_train,
    y_train_5,
    cv=3)

print(precision_score(y_train_5, y_train_pred_forest))
print(recall_score(y_train_5, y_train_pred_forest))
```

```
0.992589481683
0.985567461185
0.831396421324
```

Multiclass Classification

```
# some algorithms (RF, Bayes, ..) can handle multiple classes
# others (SVMs, linear, ...) cannot

# one-vs-all (OVA) strategy for 0-9 digit classication:
# 10 binary classifiers, one for each digit -- select class with
 highest score

# one-vs-one (OVO) strategy:
# train classifiers for every PAIR of digits -- N*(N-1)/2 classi
fiers needed!

# Scikit detects using binary classifier when multi-class proble
m is present,
# auto-selects OVA.

sgd_clf.fit(X_train, y_train)
print(sgd_clf.predict([some_digit])) # can SGD correctly predict
 the "five"?
```

```
[ 5.]
```

```
# let's see 10 scores, one per class.
# highest score corresponds to "five".

some_digit_scores = sgd_clf.decision_function([some_digit])
print(some_digit_scores)
print(sgd_clf.classes_)
```

```
[[-177277.32782496 -561668.18573184 -385895.43788059 -114677.953
60751
  -410210.58824666   57844.42736708 -654717.63929413 -200777.651
0135
  -772154.70175904 -614737.18986655]]
[ 0.  1.  2.  3.  4.  5.  6.  7.  8.  9.]
```

```
# to force Scikit to use OVO (in this case) or OVA: use correspo
nding classifier.

from sklearn.multiclass import OneVsOneClassifier

ovo_clf = OneVsOneClassifier(SGDClassifier(random_state=42))
ovo_clf.fit(X_train, y_train)
print("prediction:\n",ovo_clf.predict([some_digit]))

# same thing for Random Forest (RF can directly handle multiple
classifications)
forest_clf.fit(X_train, y_train)
print("prediction via Random Forest:\n",forest_clf.predict([some
_digit]))
print("probability via Random Forest:\n",forest_clf.predict_prob
a([some_digit]))
```

```
prediction:
 [ 5.]
prediction via Random Forest:
 [ 5.]
probability via Random Forest:
 [[ 0.1  0.   0.   0.1  0.   0.8  0.   0.   0.   0. ]]
```

```
# let's check these classifiers via CV. SGD first.

print("CV score:\n",cross_val_score(
    sgd_clf,
    X_train,
    y_train,
    cv=3,
    scoring="accuracy"))
```

```
CV score:
 [ 0.84843031  0.85419271  0.81062159]
```

```
# scaling the inputs should help improve the scores.

from sklearn.preprocessing import StandardScaler
scaler = StandardScaler()

X_train_scaled = scaler.fit_transform(X_train.astype(np.float64)
)

print("CV score, scaled inputs:\n",cross_val_score(
    sgd_clf,
    X_train_scaled,
    y_train,
    cv=3,
    scoring="accuracy"))
```

```
CV score, scaled inputs:
 [ 0.91011798  0.91089554  0.90908636]
```

Error Analysis

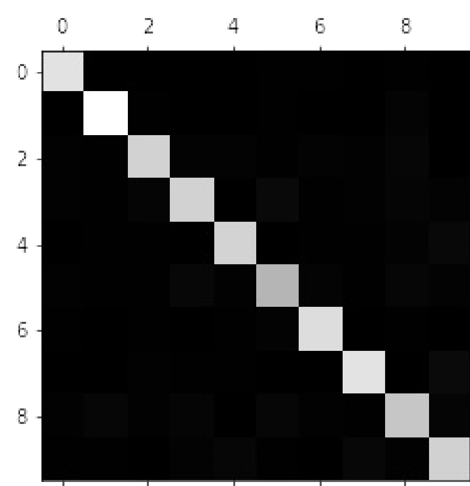
```
# as earlier: a confusion matrix from the SGD classificer

y_train_pred = cross_val_predict(
    sgd_clf,
    X_train_scaled,
    y_train,
    cv=3)

conf_mx = confusion_matrix(y_train, y_train_pred)
print("confusion matrix:\n",conf_mx)

# image equivalent
plt.matshow(conf_mx, cmap=plt.cm.gray)
plt.show()
```

```
confusion matrix:
 [[5735    4   24   11   13   45   43    8   37    3]
 [   1 6489   43   24    6   35    8    8  116   12]
 [  57   38 5329   88   79   27   92   60  174   14]
 [  53   41  140 5333    2  234   35   60  142   91]
 [  17   26   36   10 5371    8   48   30   77  219]
 [  69   38   39  185   76 4600  114   28  175   97]
 [  34   24   42    2   41   95 5625    7   48    0]
 [  22   21   64   31   49    9    8 5792   14  255]
 [  54  157   70  148   14  158   58   28 5029  135]
 [  42   37   25   85  155   36    2  193   75 5299]]
```



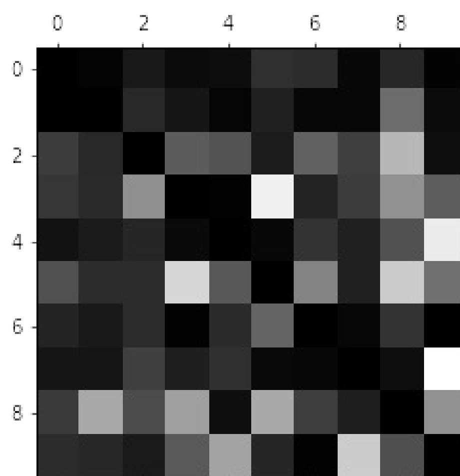
```
# focus on errors.
# 1st: divide each value in confusion matrix by #images in corre
sponding class
# (compares error rates instead of #errors)

row_sums = conf_mx.sum(axis=1, keepdims=True)
norm_conf_mx = conf_mx / row_sums

# fill diagonals with zeroes to keep only the errors, and plot.
# brighter colors = more misclassifications

np.fill_diagonal(norm_conf_mx, 0)
plt.matshow(norm_conf_mx, cmap=plt.cm.gray)
plt.show()

# rows = actual classes
# cols = predicted classes
# 8s & 9s are a problem.
```



```
# more on analyzing individual errors

# EXTRA
def plot_digits(instances, images_per_row=10, **options):
    size = 28
    images_per_row = min(len(instances), images_per_row)
    images = [instance.reshape(size,size) for instance in instan
ces]
    n_rows = (len(instances) - 1) // images_per_row + 1
    row_images = []
    n_empty = n_rows * images_per_row - len(instances)
    images.append(np.zeros((size, size * n_empty)))
    for row in range(n_rows):
        rimages = images[row * images_per_row : (row + 1) * imag
es_per_row]
        row_images.append(np.concatenate(rimages, axis=1))
    image = np.concatenate(row_images, axis=0)
    plt.imshow(image, cmap = matplotlib.cm.binary, **options)
    plt.axis("off")

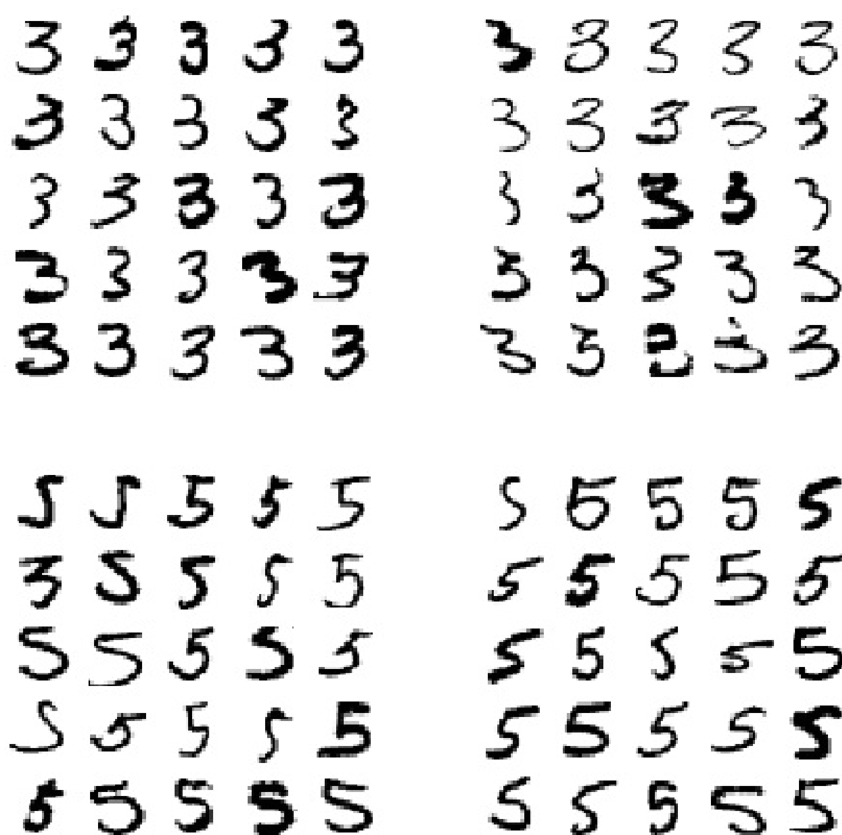
cl_a, cl_b = 3, 5

X_aa = X_train[(y_train == cl_a) & (y_train_pred == cl_a)]
X_ab = X_train[(y_train == cl_a) & (y_train_pred == cl_b)]
X_ba = X_train[(y_train == cl_b) & (y_train_pred == cl_a)]
X_bb = X_train[(y_train == cl_b) & (y_train_pred == cl_b)]

plt.figure(figsize=(8,8))
plt.subplot(221); plot_digits(X_aa[:25], images_per_row=5)
plt.subplot(222); plot_digits(X_ab[:25], images_per_row=5)
plt.subplot(223); plot_digits(X_ba[:25], images_per_row=5)
plt.subplot(224); plot_digits(X_bb[:25], images_per_row=5)
plt.show()

# shows difficulty in seeing difference between threes and fives.

# We used SGDclassifier, which is sensitive to image shifts/rota
tes.
```



MultiLabel Classification

```
# use case: returning multiple classes for each instance
# (example: multiple people's faces in one picture.)

# create y_multilabel array with 2 target labels for each digit
image:
# first = large digit (7,8,9)?; second = odd (1,3,5,7,9)?


from sklearn.neighbors import KNeighborsClassifier

y_train_large = (y_train >= 7)
y_train_odd = (y_train % 2 == 1)

print("large nums?\n",y_train_large)
print("odd nums?\n",y_train_odd)

y_multilabel = np.c_[y_train_large, y_train_odd]

print("combined (multilabel)?\n",y_multilabel)

# KNeighbors classifier supports multilabeling

knn_clf = KNeighborsClassifier()
knn_clf.fit(X_train, y_multilabel)

# make example prediction using "some_digit" from above
# >= 7 = false (correct); odd digit = true (correct)

print("KNN prediction of some_digit: (>=7? odd?)\n",knn_clf.pred
ict([some_digit]))
```

```
large nums?
 [False False  True ..., False False False]
odd nums?
 [False False  True ..., False False False]
combined (multilabel)?
 [[False False]
 [False False]
 [ True  True]
 ...,
 [False False]
 [False False]
 [False False]]
KNN prediction of some_digit: (>=7? odd?)
 [[False  True]]
```

```
# another example: find avg F1 score across all labels

y_train_knn_pred = cross_val_predict(knn_clf, X_train, y_train,
cv=3)

print(f1_score(
    y_train,
    y_train_knn_pred,
    average="macro")) # use "weighted" if more weight to be give
n to more common labels.
```

```
0.968186511757
```

MultiOutput Classification

```
# generalization of multilabel, where each label can have multip
le values.
# example: build image noise removal system

# start by adding noise to MNIST dataset

import numpy.random as rnd

noise = rnd.randint(0, 100, (len(X_train), 784))
X_train_mod = X_train + noise
noise = rnd.randint(0, 100, (len(X_test), 784))
X_test_mod = X_test + noise

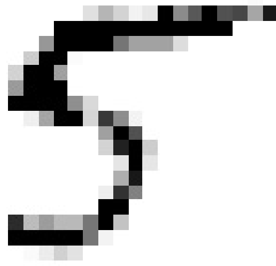
y_train_mod = X_train
y_test_mod = X_test

some_index = 5500
```

```
def plot_digit(data):
    image = data.reshape(28, 28)
    plt.imshow(image, cmap = matplotlib.cm.binary,
               interpolation="nearest")
    plt.axis("off")
```

```
# train classifier, and clean up the image
knn_clf.fit(X_train_mod, y_train_mod)
clean_digit = knn_clf.predict([X_test_mod[some_index]])

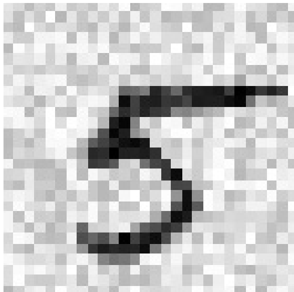
plot_digit(clean_digit)
```

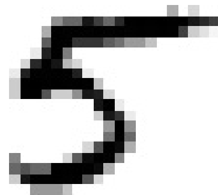


```
some_index = 5500

plt.subplot(121); plot_digit(X_test_mod[some_index])
plt.subplot(122); plot_digit(y_test_mod[some_index])
#save_fig("noisy_digit_example_plot")
plt.show()

# left: noisy image; right: cleaned up
```





Training Models - Intro

Linear Regression

- y = theta0 + (theta1 *x1) + (theta2* x2) + ...
- = h(theta)(x)
- = theta^T (dot) x --- theta^T = theta vector, transposed (row instead of col)

- Training a model = finding theta that minimizes error function (ex: MSE)

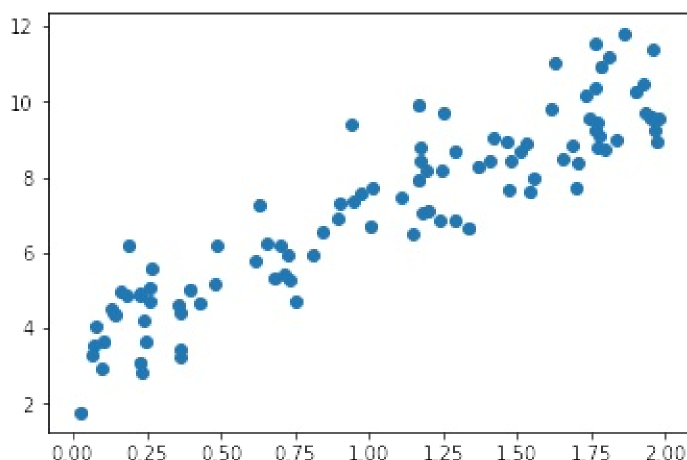


Normal Equation: finds theta that minimizes cost function

```
# generate some data
import numpy as np
X = 2 * np.random.rand(100, 1)
y = 4 + 3 * X + np.random.randn(100, 1)

%matplotlib inline
import matplotlib.pyplot as plt

plt.scatter(X,y)
plt.show()
```



```
# find theta.
# 1) use NumPy's matrix inverse function.
# 2) use dot method for matrix multiply.

X_b = np.c_[np.ones((100, 1)), X] # add x0 = 1 to each instance
theta_best = np.linalg.inv(X_b.T.dot(X_b)).dot(X_b.T).dot(y)

# results:
print(theta_best) # compare to generated data: y = 4 + 3x + noise
```

```
[[ 3.58859665]
 [ 3.41876053]]
```

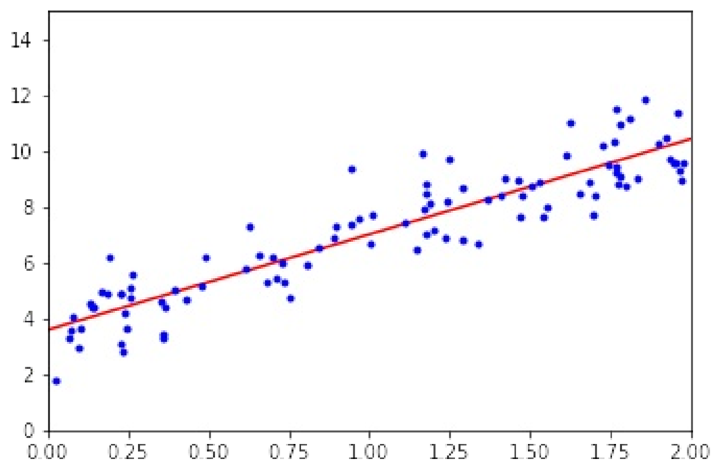
```
# make some predictions

X_new     = np.array([[0],[1],[2]])
X_new_b   = np.c_[np.ones((3, 1)), X_new] # add x0 = 1 to each i
nstance
y_predict = X_new_b.dot(theta_best)
print(y_predict)

# then plot

plt.plot(X_new, y_predict, "r-")
plt.plot(X, y, "b.")
plt.axis([0, 2, 0, 15])
plt.show()
```

```
[[  3.58859665]
 [  7.00735719]
 [ 10.42611772]]
```




```
# Scikit equivalent
from sklearn.linear_model import LinearRegression
lin_reg = LinearRegression()

lin_reg.fit(X,y)
print("intercept & coefficient:\n", lin_reg.intercept_, lin_reg.
coef_)
print("predictions:\n", lin_reg.predict(X_new))
```

```
intercept & coefficient:
 [ 3.58859665] [[ 3.41876053]]
predictions:
 [[  3.58859665]
 [  7.00735719]
 [ 10.42611772]]
```

Gradient Descent

```
# Gradient Descent - Batch
# (Batch: math includes full training set X.)

# need to find partial derivative (slope) of the cost function
# for each model parameter (theta).

theta_path_bgd = []

eta = 0.1 # learning rate
n_iterations = 1000
m = 100

theta = np.random.randn(2,1) # random initialization

for iteration in range(n_iterations):
    gradients = 2/m * X_b.T.dot(X_b.dot(theta) - y)
    theta = theta - eta * gradients
    theta_path_bgd.append(theta)

print(theta)
```

```
[[ 3.58859665]
 [ 3.41876053]]
```

```
# Gradient Descent - Stochastic
# Stochastic: finds gradients based on random instances
# adv: better for huge datasets
# dis: much more erratic than batch GD
#       -- good for avoiding local minima
#       -- bad b/c may not find optimum sol'n

# simulated annealing helps. (gradually reduces learning rate)

theta_path_sgd = []

n_epochs, t0, t1 = 50, 5, 50 # learning schedule hyperparameters

def learning_schedule(t):
    return t0 / (t + t1)

theta = np.random.randn(2,1) # random initialization

for epoch in range(n_epochs):
    for i in range(m):
        random_index = np.random.randint(m)
        xi = X_b[random_index:random_index+1]
        yi =   y[random_index:random_index+1]

        gradients = 2 * xi.T.dot(xi.dot(theta) - yi)

        eta = learning_schedule(epoch * m + i)
        theta = theta - eta * gradients
        theta_path_sgd.append(theta)

print(theta)
```

```
[[ 3.6036273 ]
 [ 3.44079196]]
```

```
# SGD Regression using Scikit:

from sklearn.linear_model import SGDRegressor

sgd_reg = SGDRegressor(n_iter=50, penalty=None, eta0=0.1)
sgd_reg.fit(X, y.ravel())

print(sgd_reg.intercept_, sgd_reg.coef_)
```

```
[ 3.57214013] [ 3.39609675]
```

```
# Gradient Descent - MiniBatch
# adv: performance boost via GPUs

theta_path_mgd = []

n_iterations = 50
minibatch_size = 20

import numpy.random as rnd

rnd.seed(42)
theta = rnd.randn(2,1)  # random initialization

t0, t1 = 10, 1000
def learning_schedule(t):
    return t0 / (t + t1)

t = 0
for epoch in range(n_iterations):
    shuffled_indices = rnd.permutation(m)
    X_b_shuffled = X_b[shuffled_indices]
    y_shuffled = y[shuffled_indices]

    for i in range(0, m, minibatch_size):
        t += 1

        xi = X_b_shuffled[i:i+minibatch_size]
        yi =   y_shuffled[i:i+minibatch_size]

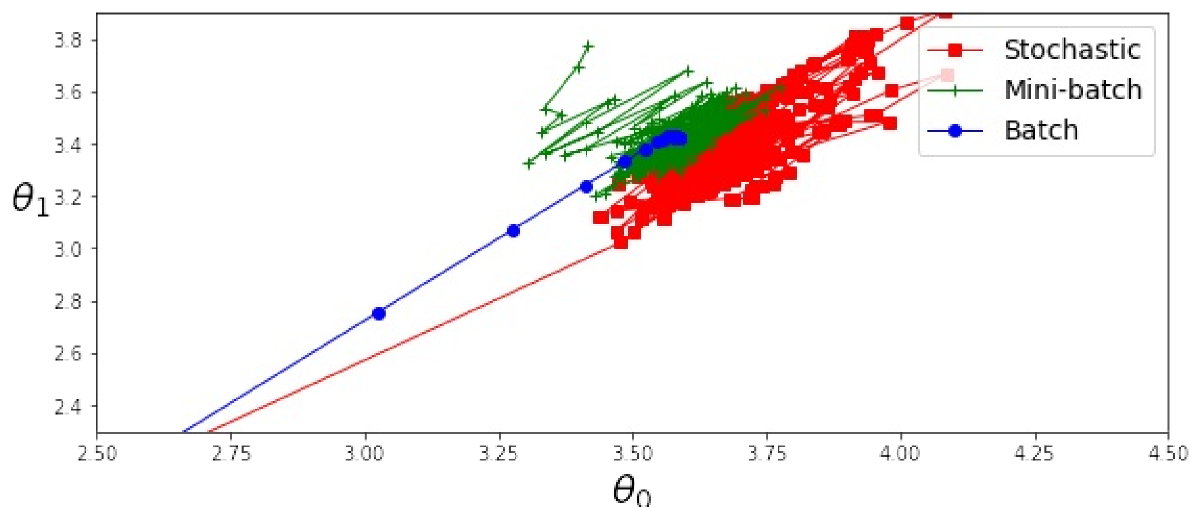
        gradients = 2 * xi.T.dot(xi.dot(theta) - yi)
        eta = learning_schedule(t)
        theta = theta - eta * gradients
        theta_path_mgd.append(theta)

print(theta)
```

```
[[ 3.70412445]
 [ 3.54124923]]
```

```
theta_path_bgd = np.array(theta_path_bgd)
theta_path_sgd = np.array(theta_path_sgd)
theta_path_mgd = np.array(theta_path_mgd)
```

```
plt.figure(figsize=(10,4))
plt.plot(theta_path_sgd[:, 0], theta_path_sgd[:, 1], "r-s", line
width=1, label="Stochastic")
plt.plot(theta_path_mgd[:, 0], theta_path_mgd[:, 1], "g-+", line
width=1, label="Mini-batch")
plt.plot(theta_path_bgd[:, 0], theta_path_bgd[:, 1], "b-o", line
width=1, label="Batch")
plt.legend(loc="upper right", fontsize=14)
plt.xlabel(r"$\theta_0$", fontsize=20)
plt.ylabel(r"$\theta_1$   ", fontsize=20, rotation=0)
plt.axis([2.5, 4.5, 2.3, 3.9])
#save_fig("gradient_descent_paths_plot")
plt.show()
```

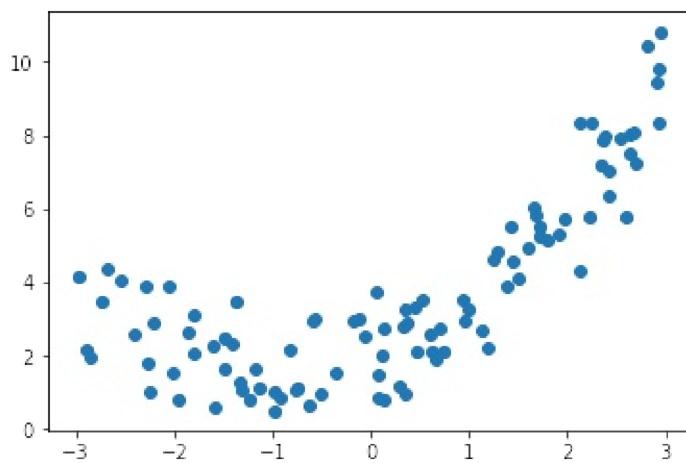


Polynomial Regression

```
# example quadratic equation + noise: y = 0.5*X^2 + X + 2 + noise

m = 100
X = 6 * np.random.rand(m, 1) - 3
y = 0.5 * X**2 + X + 2 + np.random.randn(m, 1)

plt.scatter(X,y)
plt.show()
```



```
# fit using Scikit

from sklearn.preprocessing import PolynomialFeatures
from sklearn.linear_model  import LinearRegression

# caution: PolynomialFeatures converts array of n features
# into array of (n+d)!/d!n! features -- combinatorial explosions
 possible :-)

poly_features = PolynomialFeatures(degree=2, include_bias=False)
print(poly_features)

# X_poly: original feature of X, plus its square.
X_poly = poly_features.fit_transform(X)

#print(X, X_poly)
print(X[0], X_poly[0])

# fit it:
lin_reg = LinearRegression()
lin_reg.fit(X_poly, y)
print(lin_reg.intercept_, lin_reg.coef_)

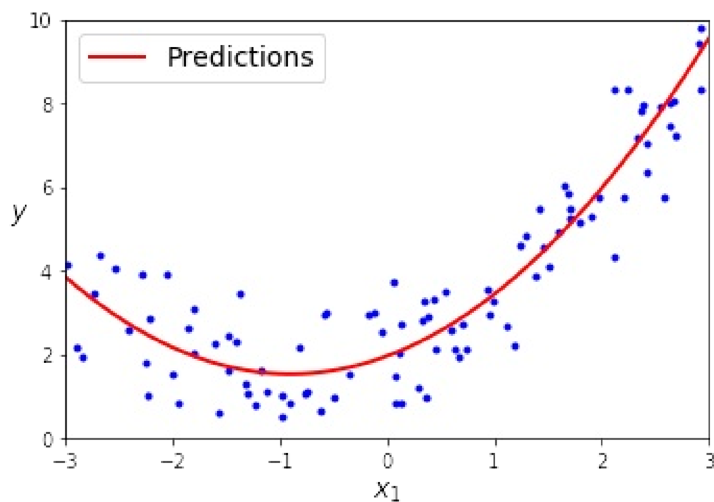
# result estimate: 0.48x(1)^2 + 0.99x(2) + 2.06
# original:        0.50x(1)^2 + 1.00x(2) + 2.00 + gaussian noise
```

```
PolynomialFeatures(degree=2, include_bias=False, interaction_onl
y=False)
[ 2.38942838] [ 2.38942838  5.709368  ]
[ 1.9735233] [[ 0.95038538  0.52577032]]
```

```
X_new      = np.linspace(-3, 3, 100).reshape(100, 1)
X_new_poly = poly_features.transform(X_new)
y_new      = lin_reg.predict(X_new_poly)

#testme = np.linspace(-3,3,20)
#print(testme, testme.reshape(20,1))

plt.plot(X, y, "b.")
plt.plot(X_new, y_new, "r-", linewidth=2, label="Predictions")
plt.xlabel("$x_1$", fontsize=14)
plt.ylabel("$y$", rotation=0, fontsize=14)
plt.legend(loc="upper left", fontsize=14)
plt.axis([-3, 3, 0, 10])
#save_fig("quadratic_predictions_plot")
plt.show()
```



Learning Curves

```
# another way to check for underfit & overfit:
# use learning curve plots to see performance vs training set si
ze.

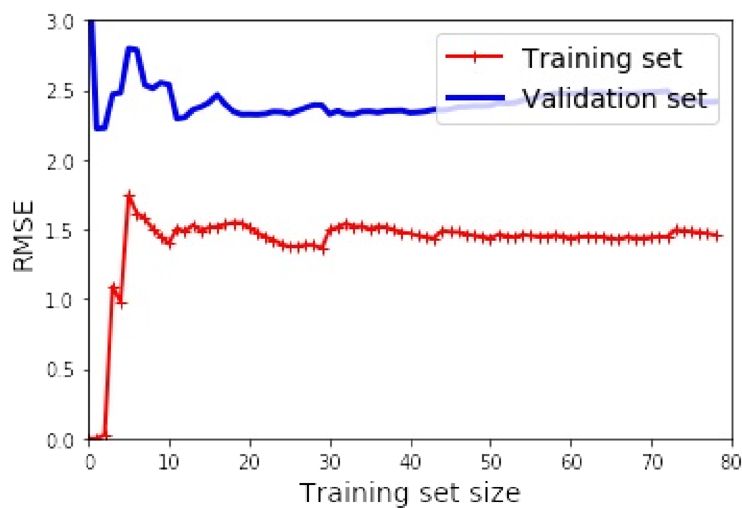
from sklearn.metrics import mean_squared_error
from sklearn.model_selection import train_test_split

# train model multiple times on various training subsets (of var
ious sizes)

def plot_learning_curves(model, X, y):
    X_train, X_val, y_train, y_val = train_test_split(X, y, test
_size=0.2, random_state=10)
    train_errors, val_errors = [], []
    for m in range(1, len(X_train)):
        model.fit(X_train[:m], y_train[:m])
        y_train_predict = model.predict(X_train[:m])
        y_val_predict = model.predict(X_val)
        train_errors.append(mean_squared_error(y_train_predict,
y_train[:m]))
        val_errors.append(mean_squared_error(y_val_predict, y_va
l))

    plt.plot(np.sqrt(train_errors), "r-+", linewidth=2, label="T
raining set")
    plt.plot(np.sqrt(val_errors), "b-", linewidth=3, label="Vali
dation set")
    plt.legend(loc="upper right", fontsize=14)
    plt.xlabel("Training set size", fontsize=14)
    plt.ylabel("RMSE", fontsize=14)

lin_reg = LinearRegression()
plot_learning_curves(lin_reg, X, y)
plt.axis([0, 80, 0, 3])
#save_fig("underfitting_learning_curves_plot")
plt.show()
```



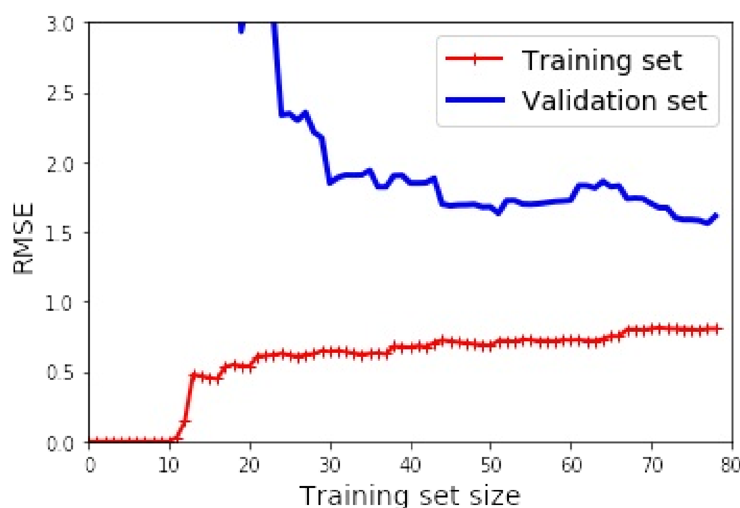

```
# repeat exercise for 10th-degree polynomial

from sklearn.pipeline import Pipeline

polynomial_regression = Pipeline((
    ("poly_features", PolynomialFeatures(degree=10, include_bias=
False)),
    ("sgd_reg", LinearRegression()),
))

plot_learning_curves(polynomial_regression, X, y)
plt.axis([0,80,0,3])
plt.show()

# note: training error rate much lower than on Linear Regression
# note: training/validation gap closes to zero. good fit?
```



Bias/Variance Tradeoff

- Bias: the part of generalization error due to wrong assumptions.
- Variance: due to model sensitivity to small training variations. (More common in high-dimensional models.)
- Irreducibility: due to data noise.

- Rule of thumb: increasing model complexity increases variance & reduces bias (and vice versa.)

Regularization

- Used to reduce overfit by constraining the model (ex: reducing the # of degrees in a polynomial).

```
# Ridge -- regularization term added to cost function.
# alpha param -- forces model weights to minimal values. higher
alpha = "flatter" function (converge to mean)
```

- Cost function: $J(\theta) = \text{MSE}(\theta) + \alpha\frac{1}{2}\sum_{i=1}^{n}\theta_i^2$

```
# build dataset

import numpy.random as rnd

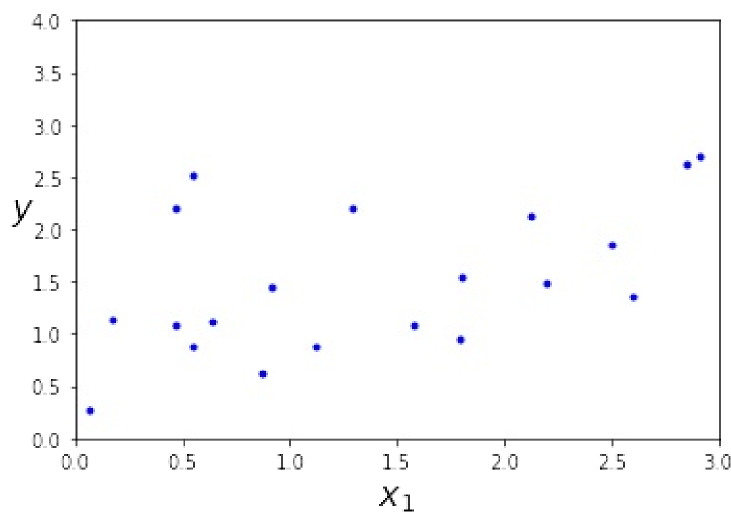
rnd.seed(42)
m = 20
X = 3 * rnd.rand(m, 1)
y = 1 + 0.5 * X + rnd.randn(m, 1) / 1.5
X_new = np.linspace(0, 3, 100).reshape(100, 1)

# plot it
plt.plot(X, y, "b.")
plt.xlabel("$x_1$", fontsize=18)
plt.ylabel("$y$", rotation=0, fontsize=18)
plt.axis([0, 3, 0, 4])

# apply Ridge regression
from sklearn.linear_model import Ridge

ridge_reg = Ridge(alpha=1, solver="cholesky")
ridge_reg.fit(X,y)
ridge_reg.predict([[0.0],[1.5],[2.0],[3.0]])
```

```
array([[ 1.00650911],
       [ 1.55071465],
       [ 1.73211649],
       [ 2.09492018]])
```




```
# Ridge using SGD:
sgd_reg = SGDRegressor(penalty="l2")
sgd_reg.fit(X,y.ravel())
ridge_reg.predict([[0.0],[1.5],[2.0],[3.0]])
```

```
array([[ 1.00650911],
       [ 1.55071465],
       [ 1.73211649],
       [ 2.09492018]])
```

```
# Lasso -- similar to Ridge, also adds regularization term
# uses L1 norm (instead of 1/2 square of L2 norm, as in Ridge.)
# -- tends to force least important features to zero.

from sklearn.linear_model import Lasso

lasso_reg = Lasso(alpha=0.1)
lasso_reg.fit(X,y)
lasso_reg.predict([[0.0],[1.5],[2.0],[3.0]])
```

```
array([ 1.14537356,  1.53788174,  1.66871781,  1.93038993])
```

```
# Elastic Net -- midddle ground.
# regularization = mix of Ridge & Lasso (mix ratio "r")

from sklearn.linear_model import ElasticNet

elastic_net = ElasticNet(alpha=0.1, l1_ratio=0.5)
elastic_net.fit(X,y)
elastic_net.predict([[0.0],[1.5],[2.0],[3.0]])
```

```
array([ 1.08639303,  1.54333232,  1.69564542,  2.00027161])
```

```
# Early Stopping -- stop training when minimum validation error
reached

# build dataset
rnd.seed(42)
m = 100
X = 6 * rnd.rand(m, 1) - 3
y = 2 + X + 0.5 * X**2 + rnd.randn(m, 1)

X_train, X_val, y_train, y_val = train_test_split(X[:50], y[:50]
.ravel(), test_size=0.5, random_state=10)

from sklearn.preprocessing import StandardScaler
from sklearn.pipeline import Pipeline

poly_scaler = Pipeline((
    ("poly_features", PolynomialFeatures(
        degree=90,
        include_bias=False)),
    ("std_scaler", StandardScaler()),
    ))

X_train_poly_scaled = poly_scaler.fit_transform(X_train)
X_val_poly_scaled   = poly_scaler.transform(X_val)

sgd_reg = SGDRegressor(n_iter=1,
```

```
                       penalty=None,
                       eta0=0.0005,
                       warm_start=True,
                       learning_rate="constant",
                       random_state=42)

n_epochs = 500
train_errors, val_errors = [], []

for epoch in range(n_epochs):
    sgd_reg.fit(X_train_poly_scaled, y_train)

    y_train_predict = sgd_reg.predict(X_train_poly_scaled)
    y_val_predict   = sgd_reg.predict(X_val_poly_scaled)

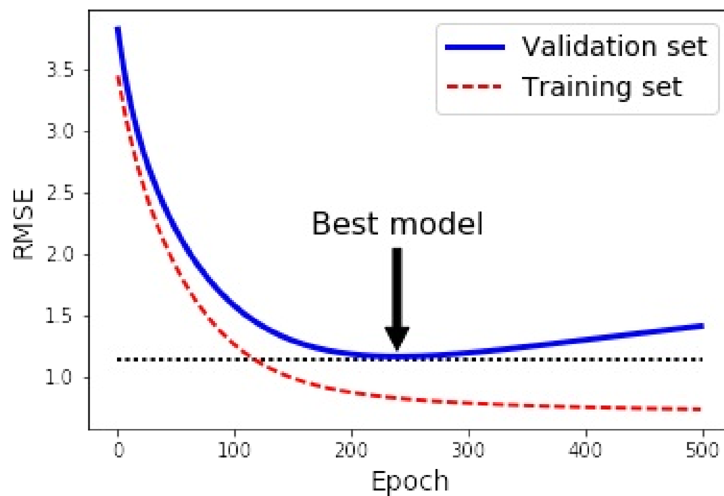
    train_errors.append(mean_squared_error(y_train_predict, y_tr
ain))
    val_errors.append(mean_squared_error(y_val_predict, y_val))

best_epoch    = np.argmin(val_errors)
best_val_rmse = np.sqrt(val_errors[best_epoch])

plt.annotate('Best model',
             xy=(best_epoch, best_val_rmse),
             xytext=(best_epoch, best_val_rmse + 1),
             ha="center",
             arrowprops=dict(facecolor='black', shrink=0.05),
             fontsize=16,
            )

best_val_rmse -= 0.03  # just to make the graph look better
plt.plot([0, n_epochs], [best_val_rmse, best_val_rmse], "k:", li
newidth=2)
plt.plot(np.sqrt(val_errors), "b-", linewidth=3, label="Validati
on set")
plt.plot(np.sqrt(train_errors), "r--", linewidth=2, label="Train
ing set")
plt.legend(loc="upper right", fontsize=14)
plt.xlabel("Epoch", fontsize=14)
plt.ylabel("RMSE", fontsize=14)
```

```
#save_fig("early_stopping_plot")
plt.show()
```



Logistic Regression

- commonly used to est probability of instance belonging to specified class. positive if >50% (labeled "1"), otherwise labeled "0".

- logistic is a sigmoid function, outputs 0<n<1.

- cost function = average over all training data. It is convex, so gradient descent will find global minimum.

```
#from sklearn import datasets
#iris = datasets.load_iris()

import numpy as np

from sklearn import datasets
iris = datasets.load_iris()
print(iris.keys())

X = iris["data"][:, 3:] # petal width
y = (iris["target"] == 2).astype(np.int) # 1 if Iris-Virginica,
else 0
```

```
dict_keys(['target_names', 'DESCR', 'data', 'target', 'feature_n
ames'])
```

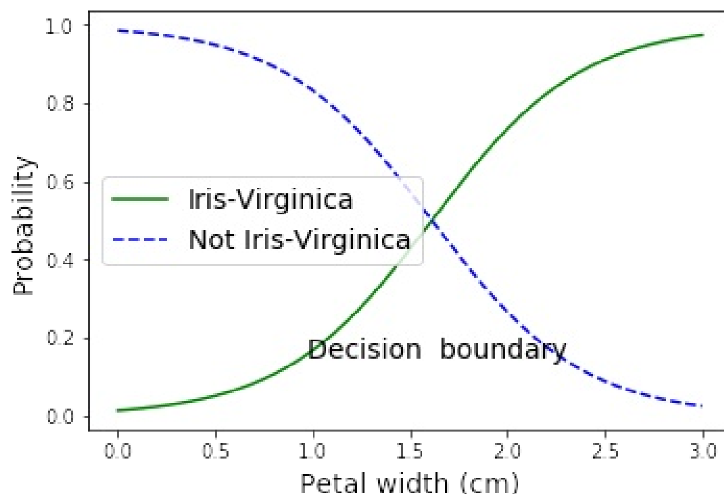
```
# train a LR model

from sklearn.linear_model import LogisticRegression
log_reg=LogisticRegression()
log_reg.fit(X,y)

# predict probability of flowers with petal widths = 0-3cm
X_new             = np.linspace(0, 3, 1000).reshape(-1, 1)
y_proba           = log_reg.predict_proba(X_new)
print(y_proba)
decision_boundary = X_new[y_proba[:, 1] >= 0.5][0]

plt.plot(X_new, y_proba[:, 1], "g-", label="Iris-Virginica")
plt.plot(X_new, y_proba[:, 0], "b--", label="Not Iris-Virginica"
)
plt.text(decision_boundary+0.02, 0.15, "Decision  boundary", fon
tsize=14, color="k", ha="center")
plt.xlabel("Petal width (cm)", fontsize=14)
plt.ylabel("Probability", fontsize=14)
plt.legend(loc="center left", fontsize=14)
plt.show()
```

```
[[ 0.98552764  0.01447236]
 [ 0.98541511  0.01458489]
 [ 0.98530171  0.01469829]
 ...,
 [ 0.02620686  0.97379314]
 [ 0.02600703  0.97399297]
 [ 0.02580868  0.97419132]]
```




```
# what's the prediction for petal length = 1.5 or 1.7cm?
print(log_reg.predict([[1.5], [1.7]]))
```

```
[0 1]
```

```
# Logistic Regressin contour plot
# with multiple decision boundaries (not just 50%)

from sklearn.linear_model import LogisticRegression

X = iris["data"][:, (2, 3)]  # petal length, petal width
y = (iris["target"] == 2).astype(np.int)

log_reg = LogisticRegression(C=10**10)
log_reg.fit(X, y)

x0, x1 = np.meshgrid(
```

```
        np.linspace(2.9, 7, 500).reshape(-1, 1),
        np.linspace(0.8, 2.7, 200).reshape(-1, 1),
    )

# ravel(): return contiguous flattened array
X_new = np.c_[x0.ravel(), x1.ravel()]

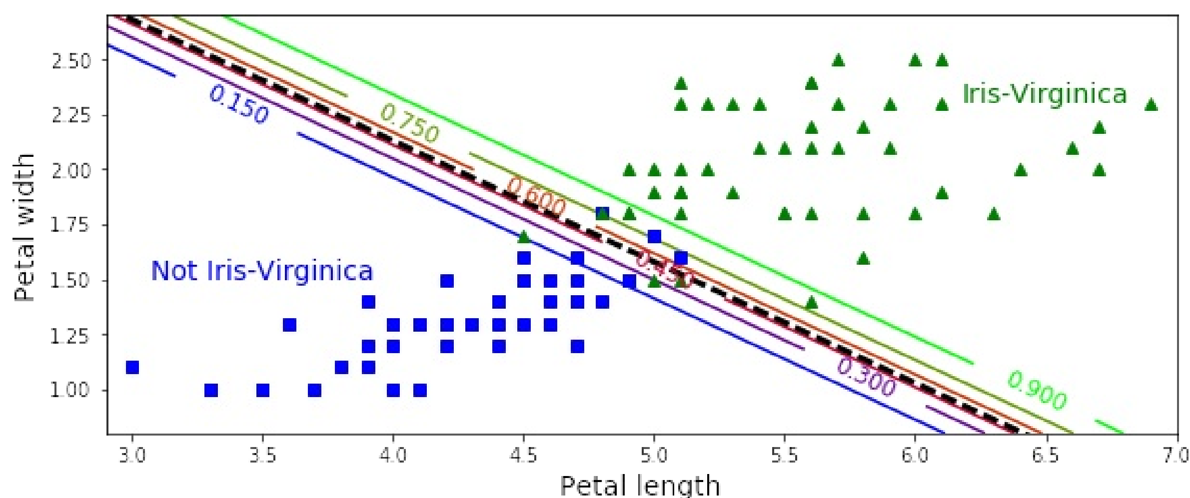
y_proba = log_reg.predict_proba(X_new)

plt.figure(figsize=(10, 4))
plt.plot(X[y==0, 0], X[y==0, 1], "bs")
plt.plot(X[y==1, 0], X[y==1, 1], "g^")

zz = y_proba[:, 1].reshape(x0.shape)
contour = plt.contour(x0, x1, zz, cmap=plt.cm.brg)


left_right = np.array([2.9, 7])
boundary = -(log_reg.coef_[0][0] * left_right + log_reg.intercep
t_[0]) / log_reg.coef_[0][1]

plt.clabel(contour, inline=1, fontsize=12)
plt.plot(left_right, boundary, "k--", linewidth=3)
plt.text(3.5, 1.5, "Not Iris-Virginica", fontsize=14, color="b",
 ha="center")
plt.text(6.5, 2.3, "Iris-Virginica", fontsize=14, color="g", ha=
"center")
plt.xlabel("Petal length", fontsize=14)
plt.ylabel("Petal width", fontsize=14)
plt.axis([2.9, 7, 0.8, 2.7])
#save_fig("logistic_regression_contour_plot")
plt.show()
```



Softmax Regression (Multinomial Logistic Regression)

- **Predicts one class at a time (multiclass, not multioutput). Use only for mutually exclusive classes.**
- Scoring for K classes: $s_k(\mathbf{x}) = \theta_k{}^T \cdot \mathbf{x}$
- Softmax function (aka normalized exponential):
- Prediction: $\hat{y} = \underset{k}{\text{argmax}}\ \sigma(\mathbf{s}(\mathbf{x}))_k = \underset{k}{\text{argmax}}\ s_k(\mathbf{x}) = \underset{k}{\text{argmax}}\ \left(\theta_k{}^T \cdot \mathbf{x}\right)$
- Uses **cross entropy** to minimize cost function. (Same as log loss, used for Logistic Regression, when k=2.) $J(\Theta) = -\frac{1}{m}\sum_{i=1}^{m}\sum_{k=1}^{K} y_k^{(i)} \log\left(\hat{p}_k^{(i)}\right)$

```
# use Softmax to classify iris flowers

X = iris["data"][:, (2, 3)] # petal length, width
y = iris["target"]

# Scikit LR can be switched to Softmax with "multinomial" settin
g.
# also defaults to L2 regularization (control with C parameter)

softmax_reg = LogisticRegression(multi_class="multinomial",solve
r="lbfgs", C=10)
softmax_reg.fit(X, y)

# predict iris 5cm long, 2cm wide:

softmax_reg.predict([[5, 2]])
softmax_reg.predict_proba([[5,2]])
```

```
array([[  6.33134078e-07,   5.75276067e-02,   9.42471760e-01]])
```

```
# softmax contour plot

x0, x1 = np.meshgrid(
        np.linspace(0, 8, 500).reshape(-1, 1),
        np.linspace(0, 3.5, 200).reshape(-1, 1),
    )
X_new = np.c_[x0.ravel(), x1.ravel()]

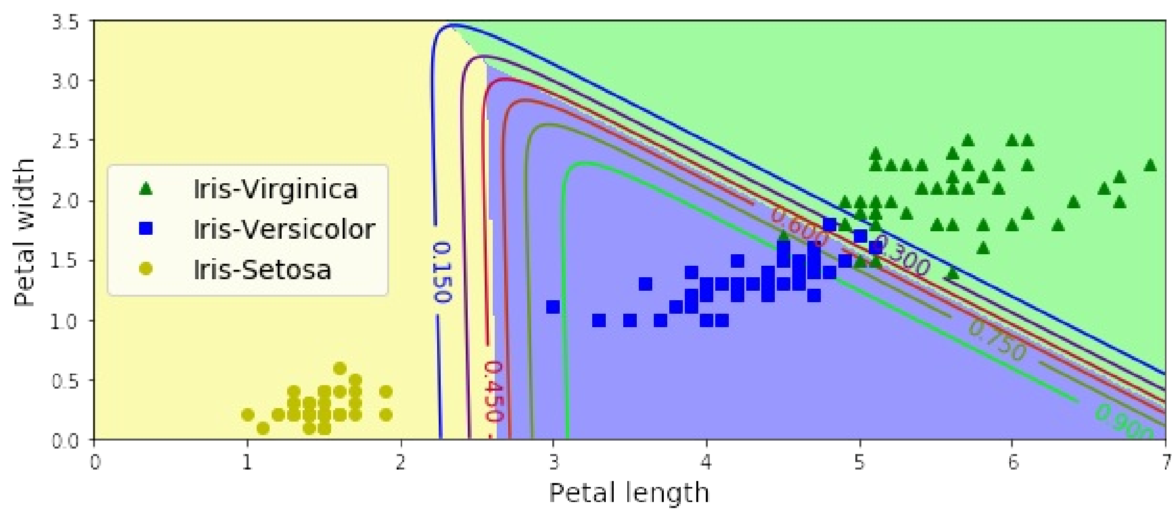

y_proba = softmax_reg.predict_proba(X_new)
y_predict = softmax_reg.predict(X_new)

zz1 = y_proba[:, 1].reshape(x0.shape)
zz = y_predict.reshape(x0.shape)

plt.figure(figsize=(10, 4))
plt.plot(X[y==2, 0], X[y==2, 1], "g^", label="Iris-Virginica")
plt.plot(X[y==1, 0], X[y==1, 1], "bs", label="Iris-Versicolor")
plt.plot(X[y==0, 0], X[y==0, 1], "yo", label="Iris-Setosa")

from matplotlib.colors import ListedColormap
custom_cmap = ListedColormap(['#fafab0','#9898ff','#a0faa0'])

plt.contourf(x0, x1, zz, cmap=custom_cmap, linewidth=5)
contour = plt.contour(x0, x1, zz1, cmap=plt.cm.brg)
plt.clabel(contour, inline=1, fontsize=12)
plt.xlabel("Petal length", fontsize=14)
plt.ylabel("Petal width", fontsize=14)
plt.legend(loc="center left", fontsize=14)
plt.axis([0, 7, 0, 3.5])
#save_fig("softmax_regression_contour_plot")
plt.show()
```


Iris-Virginica
Iris-Versicolor
Iris-Setosa
Petal width
Petal length
0.150
0.450
0.600
0.300
0.750
0.900
0
1
2
3
4
5
6
7
0.0
0.5
1.0
1.5
2.0
2.5
3.0
3.5

SVM Classification (Linear)

- Well suited for complex, small/medium dataset classification.

```
%matplotlib inline
import matplotlib.pyplot as plt
```

```
# Large margin classification:

from sklearn.svm import SVC
from sklearn import datasets

iris = datasets.load_iris()
X = iris["data"][:, (2, 3)]  # petal length, petal width
y = iris["target"]

setosa_or_versicolor = (y == 0) | (y == 1)
X = X[setosa_or_versicolor]
y = y[setosa_or_versicolor]

# SVM Classifier model
svm_clf = SVC(kernel="linear", C=float("inf"))
svm_clf.fit(X, y)
```

```
SVC(C=inf, cache_size=200, class_weight=None, coef0=0.0,
  decision_function_shape=None, degree=3, gamma='auto', kernel='
linear',
  max_iter=-1, probability=False, random_state=None, shrinking=T
rue,
  tol=0.001, verbose=False)
```

```
# Bad models

import numpy as np
```

```
x0 = np.linspace(0, 5.5, 200)
pred_1 = 5*x0 - 20
pred_2 = x0 - 1.8
pred_3 = 0.1 * x0 + 0.5

def plot_svc_decision_boundary(svm_clf, xmin, xmax):
    w = svm_clf.coef_[0]
    b = svm_clf.intercept_[0]

    # At the decision boundary, w0*x0 + w1*x1 + b = 0
    # => x1 = -w0/w1 * x0 - b/w1
    x0 = np.linspace(xmin, xmax, 200)
    decision_boundary = -w[0]/w[1] * x0 - b/w[1]

    margin = 1/w[1]
    gutter_up = decision_boundary + margin
    gutter_down = decision_boundary - margin

    svs = svm_clf.support_vectors_
    plt.scatter(svs[:, 0], svs[:, 1], s=180, facecolors='#FFAAAA'
)
    plt.plot(x0, decision_boundary, "k-",  linewidth=2)
    plt.plot(x0, gutter_up,         "k--", linewidth=2)
    plt.plot(x0, gutter_down,       "k--", linewidth=2)

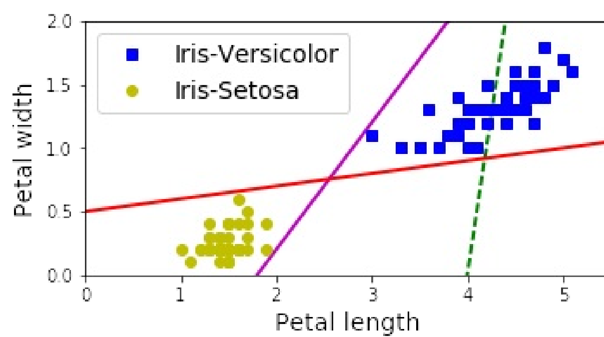
plt.figure(figsize=(12,2.7))

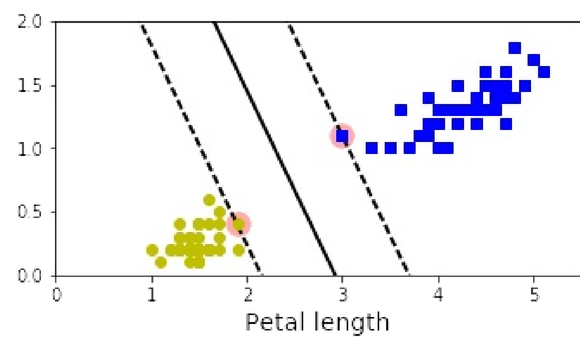
plt.subplot(121)
plt.plot(x0, pred_1, "g--", linewidth=2)
plt.plot(x0, pred_2, "m-", linewidth=2)
plt.plot(x0, pred_3, "r-", linewidth=2)
plt.plot(X[:, 0][y==1], X[:, 1][y==1], "bs", label="Iris-Versico
lor")
plt.plot(X[:, 0][y==0], X[:, 1][y==0], "yo", label="Iris-Setosa"
)
plt.xlabel("Petal length", fontsize=14)
plt.ylabel("Petal width", fontsize=14)
plt.legend(loc="upper left", fontsize=14)
plt.axis([0, 5.5, 0, 2])
```

```
plt.subplot(122)
plot_svc_decision_boundary(svm_clf, 0, 5.5)
plt.plot(X[:, 0][y==1], X[:, 1][y==1], "bs")
plt.plot(X[:, 0][y==0], X[:, 1][y==0], "yo")
plt.xlabel("Petal length", fontsize=14)
plt.axis([0, 5.5, 0, 2])
plt.show()

# On left:
# dashed line = basically useless decision boundary.
# solid lines = OK for this dataset, but no margins. Probably wi
ll not work well on new instances.

# On right: SVM finds widest possible "street" between classes.
```





```
# sensitivity to feature scaling:

Xs = np.array([[1, 50], [5, 20], [3, 80], [5, 60]]).astype(np.fl
oat64)
ys = np.array([0, 0, 1, 1])
svm_clf = SVC(kernel="linear", C=100)
svm_clf.fit(Xs, ys)

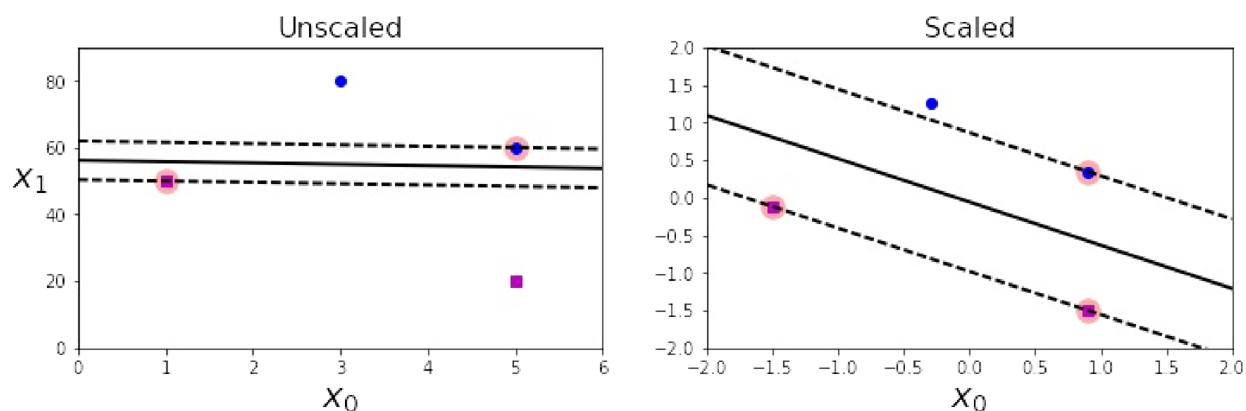
plt.figure(figsize=(12,3.2))
plt.subplot(121)
plt.plot(Xs[:, 0][ys==1], Xs[:, 1][ys==1], "bo")
plt.plot(Xs[:, 0][ys==0], Xs[:, 1][ys==0], "ms")
plot_svc_decision_boundary(svm_clf, 0, 6)
plt.xlabel("$x_0$", fontsize=20)
plt.ylabel("$x_1$  ", fontsize=20, rotation=0)
plt.title("Unscaled", fontsize=16)
plt.axis([0, 6, 0, 90])

from sklearn.preprocessing import StandardScaler
scaler = StandardScaler()
X_scaled = scaler.fit_transform(Xs)
svm_clf.fit(X_scaled, ys)

plt.subplot(122)
plt.plot(X_scaled[:, 0][ys==1], X_scaled[:, 1][ys==1], "bo")
plt.plot(X_scaled[:, 0][ys==0], X_scaled[:, 1][ys==0], "ms")
plot_svc_decision_boundary(svm_clf, -2, 2)
plt.xlabel("$x_0$", fontsize=20)
plt.title("Scaled", fontsize=16)
plt.axis([-2, 2, -2, 2])

# SVMs are sensitive to feature scaling.
# Plot on right has much more robust feature boundary.
```

```
[-2, 2, -2, 2]
```




```
X_scaled
```

```
array([[-1.50755672, -0.11547005],
       [ 0.90453403, -1.5011107 ],
       [-0.30151134,  1.27017059],
       [ 0.90453403,  0.34641016]])
```

```
# "hard" margin classification:
# - all instances need to be "out of the street".
# - all instances need to be "on the right side of the street".
# problem: doable only if data is linearly separable
# problem: very sensitive to outliers

X_outliers = np.array([[3.4, 1.3], [3.2, 0.8]])
y_outliers = np.array([0, 0])
Xo1 = np.concatenate([X, X_outliers[:1]], axis=0)
yo1 = np.concatenate([y, y_outliers[:1]], axis=0)
Xo2 = np.concatenate([X, X_outliers[1:]], axis=0)
yo2 = np.concatenate([y, y_outliers[1:]], axis=0)

svm_clf2 = SVC(kernel="linear", C=10**9)#float("inf"))
svm_clf2.fit(Xo2, yo2)

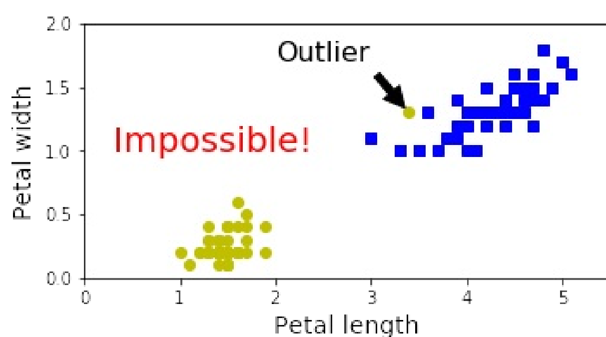
plt.figure(figsize=(12,2.7))

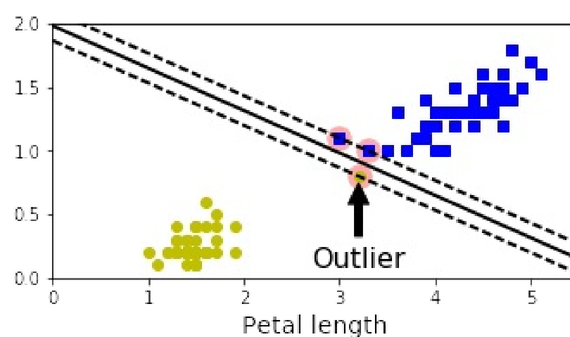
plt.subplot(121)
plt.plot(Xo1[:, 0][yo1==1], Xo1[:, 1][yo1==1], "bs")
plt.plot(Xo1[:, 0][yo1==0], Xo1[:, 1][yo1==0], "yo")
```

```
plt.text(0.3, 1.0, "Impossible!", fontsize=20, color="red")
plt.xlabel("Petal length", fontsize=14)
plt.ylabel("Petal width", fontsize=14)
plt.annotate("Outlier",
             xy=(X_outliers[0][0], X_outliers[0][1]),
             xytext=(2.5, 1.7),
             ha="center",
             arrowprops=dict(facecolor='black', shrink=0.1),
             fontsize=16,
            )
plt.axis([0, 5.5, 0, 2])

plt.subplot(122)
plt.plot(Xo2[:, 0][yo2==1], Xo2[:, 1][yo2==1], "bs")
plt.plot(Xo2[:, 0][yo2==0], Xo2[:, 1][yo2==0], "yo")
plot_svc_decision_boundary(svm_clf2, 0, 5.5)
plt.xlabel("Petal length", fontsize=14)
plt.annotate("Outlier",
             xy=(X_outliers[1][0], X_outliers[1][1]),
             xytext=(3.2, 0.08),
             ha="center",
             arrowprops=dict(facecolor='black', shrink=0.1),
             fontsize=16,
            )
plt.axis([0, 5.5, 0, 2])
```

```
[0, 5.5, 0, 2]
```






```
X_scaled
```

```
array([[-1.50755672, -0.11547005],
       [ 0.90453403, -1.5011107 ],
       [-0.30151134,  1.27017059],
       [ 0.90453403,  0.34641016]])
```

```
# soluton to "hard margins" problem:
# control hardness with C hyperparameter

from sklearn import datasets
from sklearn.pipeline import Pipeline
from sklearn.preprocessing import StandardScaler
from sklearn.svm import LinearSVC

iris = datasets.load_iris()
X = iris["data"][:, (2, 3)]  # petal length, petal width
y = (iris["target"] == 2).astype(np.float64)  # Iris-Virginica

scaler = StandardScaler()
svm_clf1 = LinearSVC(C=100, loss="hinge")
svm_clf2 = LinearSVC(C=1, loss="hinge")

scaled_svm_clf1 = Pipeline((
        ("scaler", scaler),
        ("linear_svc", svm_clf1),
    ))
scaled_svm_clf2 = Pipeline((
        ("scaler", scaler),
        ("linear_svc", svm_clf2),
    ))

scaled_svm_clf1.fit(X, y)
scaled_svm_clf2.fit(X, y)

scaled_svm_clf2.predict([[5.5, 1.7]])
```

```
array([ 1.])
```

```
X_scaled
```

```
array([[-1.50755672, -0.11547005],
       [ 0.90453403, -1.5011107 ],
       [-0.30151134,  1.27017059],
       [ 0.90453403,  0.34641016]])
```

```
# Convert to unscaled parameters
b1 = svm_clf1.decision_function([-scaler.mean_ / scaler.scale_])
b2 = svm_clf2.decision_function([-scaler.mean_ / scaler.scale_])
w1 = svm_clf1.coef_[0] / scaler.scale_
w2 = svm_clf2.coef_[0] / scaler.scale_
svm_clf1.intercept_ = np.array([b1])
svm_clf2.intercept_ = np.array([b2])
svm_clf1.coef_ = np.array([w1])
svm_clf2.coef_ = np.array([w2])

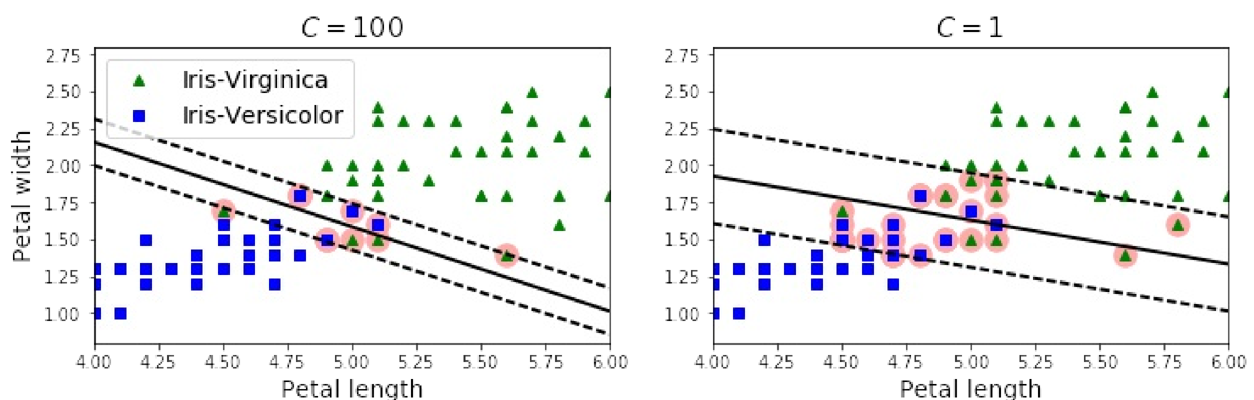
# Find support vectors (LinearSVC does not do this automatically)

t = y * 2 - 1
support_vectors_idx1 = (t * (X.dot(w1) + b1) < 1).ravel()
support_vectors_idx2 = (t * (X.dot(w2) + b2) < 1).ravel()
svm_clf1.support_vectors_ = X[support_vectors_idx1]
svm_clf2.support_vectors_ = X[support_vectors_idx2]
```

```
plt.figure(figsize=(12,3.2))
plt.subplot(121)
plt.plot(X[:, 0][y==1], X[:, 1][y==1], "g^", label="Iris-Virgini
ca")
plt.plot(X[:, 0][y==0], X[:, 1][y==0], "bs", label="Iris-Versico
lor")
plot_svc_decision_boundary(svm_clf1, 4, 6)
plt.xlabel("Petal length", fontsize=14)
plt.ylabel("Petal width", fontsize=14)
plt.legend(loc="upper left", fontsize=14)
plt.title("$C = {}$".format(svm_clf1.C), fontsize=16)
plt.axis([4, 6, 0.8, 2.8])

plt.subplot(122)
plt.plot(X[:, 0][y==1], X[:, 1][y==1], "g^")
plt.plot(X[:, 0][y==0], X[:, 1][y==0], "bs")
plot_svc_decision_boundary(svm_clf2, 4, 6)
plt.xlabel("Petal length", fontsize=14)
plt.title("$C = {}$".format(svm_clf2.C), fontsize=16)
plt.axis([4, 6, 0.8, 2.8])
```

```
[4, 6, 0.8, 2.8]
```



SVM Classification (Non-Linear)

```
# some (most?) datasets are not linearly separable. simple examp
le below.
```

```
X1D = np.linspace(-4, 4, 9).reshape(-1, 1)

X2D = np.c_[X1D, X1D**2] # adds 2nd, non-linear dimension.

y = np.array([0, 0, 1, 1, 1, 1, 1, 0, 0])

plt.figure(figsize=(10, 4))

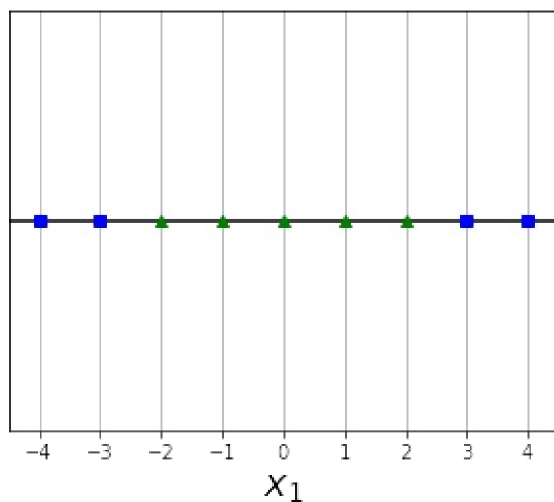
plt.subplot(121)
plt.grid(True, which='both')
plt.axhline(y=0, color='k')
plt.plot(X1D[:, 0][y==0], np.zeros(4), "bs")
plt.plot(X1D[:, 0][y==1], np.zeros(5), "g^")
plt.gca().get_yaxis().set_ticks([])
plt.xlabel(r"$x_1$", fontsize=20)
plt.axis([-4.5, 4.5, -0.2, 0.2])

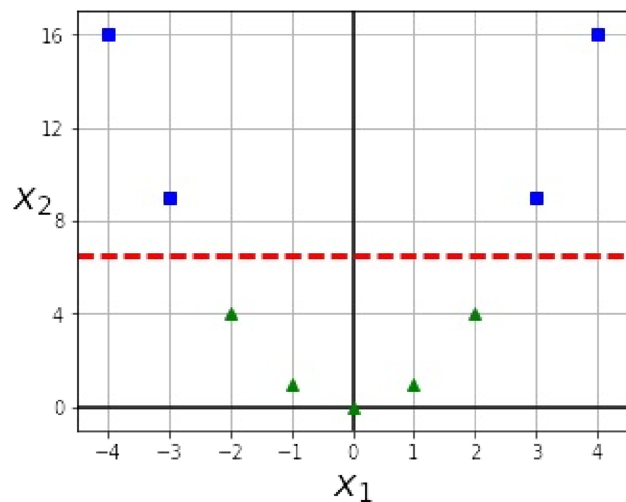
plt.subplot(122)
plt.grid(True, which='both')
plt.axhline(y=0, color='k')
plt.axvline(x=0, color='k')
plt.plot(X2D[:, 0][y==0], X2D[:, 1][y==0], "bs")
plt.plot(X2D[:, 0][y==1], X2D[:, 1][y==1], "g^")
plt.xlabel(r"$x_1$", fontsize=20)
plt.ylabel(r"$x_2$", fontsize=20, rotation=0)
plt.gca().get_yaxis().set_ticks([0, 4, 8, 12, 16])
plt.plot([-4.5, 4.5], [6.5, 6.5], "r--", linewidth=3)
plt.axis([-4.5, 4.5, -1, 17])

plt.subplots_adjust(right=1)

#save_fig("higher_dimensions_plot", tight_layout=False)
plt.show()

# result: adding 2nd dimension (on right) makes dataset linearly
 separable
```



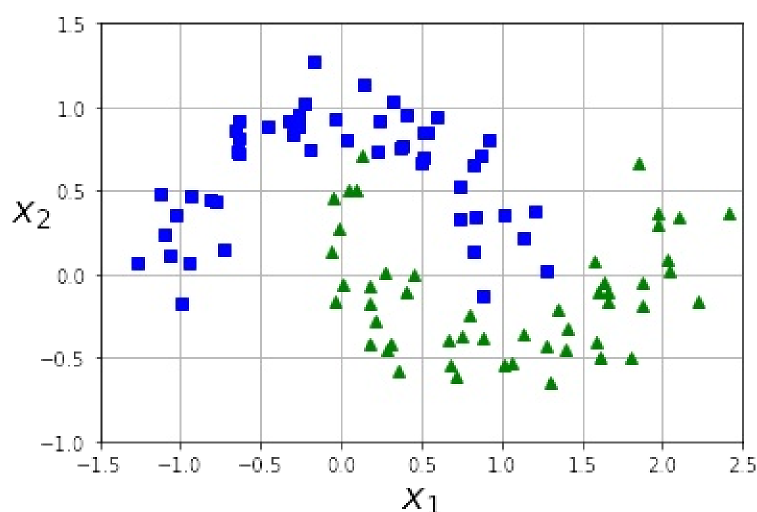



```
# test on "moons" dataset

from sklearn.datasets import make_moons
X, y = make_moons(n_samples=100, noise=0.15, random_state=42)

def plot_dataset(X, y, axes):
    plt.plot(X[:, 0][y==0], X[:, 1][y==0], "bs")
    plt.plot(X[:, 0][y==1], X[:, 1][y==1], "g^")
    plt.axis(axes)
    plt.grid(True, which='both')
    plt.xlabel(r"$x_1$", fontsize=20)
    plt.ylabel(r"$x_2$", fontsize=20, rotation=0)

plot_dataset(X, y, [-1.5, 2.5, -1, 1.5])
plt.show()
```



```
# do this in Scikit with a Pipeline. Contents:
# 1) Polynomial Features
# 2) StandardScaler
# 3) LinearSVC

from sklearn.pipeline import Pipeline
from sklearn.preprocessing import PolynomialFeatures

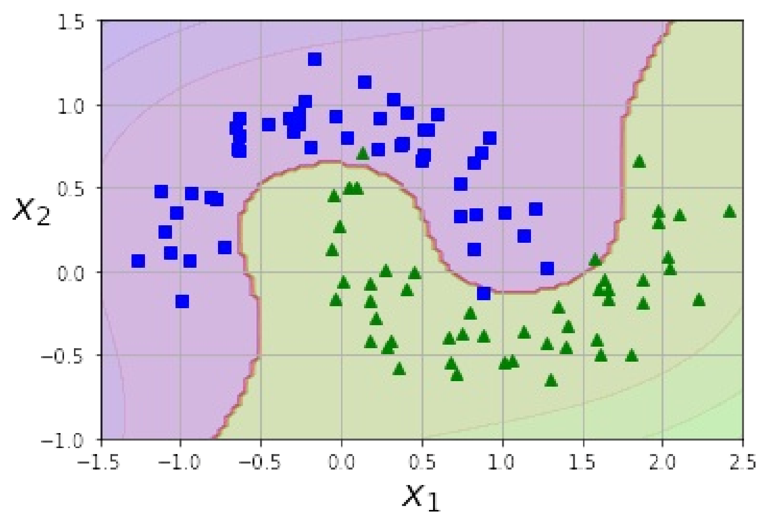
polynomial_svm_clf = Pipeline((
        ("poly_features", PolynomialFeatures(degree=3)),
        ("scaler", StandardScaler()),
        ("svm_clf", LinearSVC(C=10, loss="hinge"))
    ))

polynomial_svm_clf.fit(X, y)

def plot_predictions(clf, axes):
    x0s = np.linspace(axes[0], axes[1], 100)
    x1s = np.linspace(axes[2], axes[3], 100)
    x0, x1 = np.meshgrid(x0s, x1s)
    X = np.c_[x0.ravel(), x1.ravel()]
    y_pred = clf.predict(X).reshape(x0.shape)
    y_decision = clf.decision_function(X).reshape(x0.shape)
    plt.contourf(x0, x1, y_pred, cmap=plt.cm.brg, alpha=0.2)
    plt.contourf(x0, x1, y_decision, cmap=plt.cm.brg, alpha=0.1)

plot_predictions(polynomial_svm_clf, [-1.5, 2.5, -1, 1.5])
plot_dataset(X, y, [-1.5, 2.5, -1, 1.5])

#save_fig("moons_polynomial_svc_plot")
plt.show()
```



Solving polynomial-feature problems (aka combinatorial explosion) via the kernel trick

```
from sklearn.svm import SVC

# train SVM classifier using 3rd-degree polynomial kernel
poly_kernel_svm_clf = Pipeline((
    ("scaler", StandardScaler()),
    ("svm_clf", SVC(
        kernel="poly", degree=3, coef0=1, C=5))))

# train SVM classifier using 10th-degree polynomial kernel (for
comparison)
poly100_kernel_svm_clf = Pipeline((
        ("scaler", StandardScaler()),
        ("svm_clf", SVC(kernel="poly", degree=10, coef0=100, C=5
))
    ))

poly_kernel_svm_clf.fit(X, y)
poly100_kernel_svm_clf.fit(X, y)

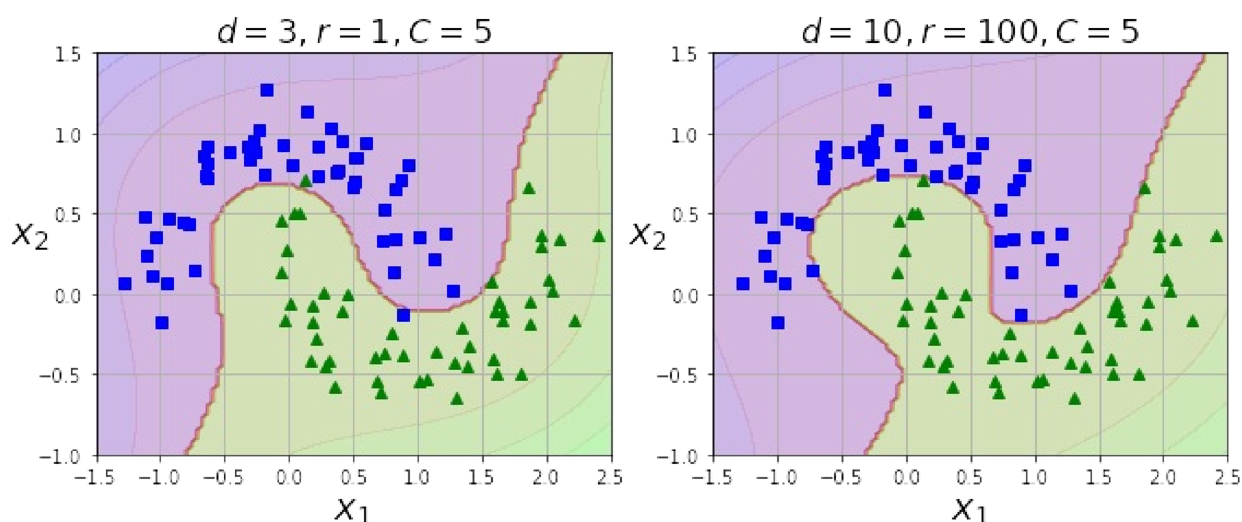
plt.figure(figsize=(11, 4))

plt.subplot(121)
plot_predictions(poly_kernel_svm_clf, [-1.5, 2.5, -1, 1.5])
plot_dataset(X, y, [-1.5, 2.5, -1, 1.5])
plt.title(r"$d=3, r=1, C=5$", fontsize=18)

plt.subplot(122)
plot_predictions(poly100_kernel_svm_clf, [-1.5, 2.5, -1, 1.5])
plot_dataset(X, y, [-1.5, 2.5, -1, 1.5])
plt.title(r"$d=10, r=100, C=5$", fontsize=18)

#save_fig("moons_kernelized_polynomial_svc_plot")
plt.show()

# left: 3rd-degree polynomial; right: 10th-degree polynomial.
# if overfitting, reduce polynomial degree. if underfitting, bum
p it up.
# "coef0": controls high- vs low-degree polynomial influence.
```



Adding Similarity Features

- **similarity function**: measures how much an instance resembles specified landmark.

```
# define similarity function to be Gaussian Radial Basis Function (RBF)
# equals 0 (far away) to 1 (at landmark)

def gaussian_rbf(x, landmark, gamma):
    return np.exp(-gamma * np.linalg.norm(x - landmark, axis=1)**2)

gamma = 0.3

x1s = np.linspace(-4.5, 4.5, 200).reshape(-1, 1)
x2s = gaussian_rbf(x1s, -2, gamma)
x3s = gaussian_rbf(x1s, 1, gamma)

XK = np.c_[gaussian_rbf(X1D, -2, gamma), gaussian_rbf(X1D, 1, gamma)]
yk = np.array([0, 0, 1, 1, 1, 1, 1, 0, 0])

plt.figure(figsize=(11, 4))

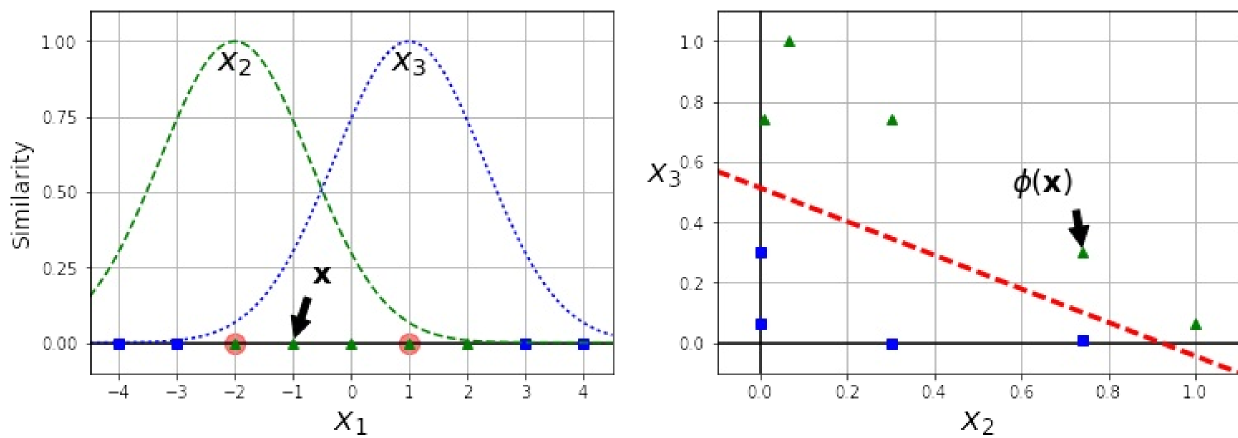
plt.subplot(121)
plt.grid(True, which='both')
```

```
plt.axhline(y=0, color='k')
plt.scatter(x=[-2, 1], y=[0, 0], s=150, alpha=0.5, c="red")
plt.plot(X1D[:, 0][yk==0], np.zeros(4), "bs")
plt.plot(X1D[:, 0][yk==1], np.zeros(5), "g^")
plt.plot(x1s, x2s, "g--")
plt.plot(x1s, x3s, "b:")
plt.gca().get_yaxis().set_ticks([0, 0.25, 0.5, 0.75, 1])
plt.xlabel(r"$x_1$", fontsize=20)
plt.ylabel(r"Similarity", fontsize=14)
plt.annotate(r'$\mathbf{x}$',
             xy=(X1D[3, 0], 0),
             xytext=(-0.5, 0.20),
             ha="center",
             arrowprops=dict(facecolor='black', shrink=0.1),
             fontsize=18,
            )
plt.text(-2, 0.9, "$x_2$", ha="center", fontsize=20)
plt.text(1, 0.9, "$x_3$", ha="center", fontsize=20)
plt.axis([-4.5, 4.5, -0.1, 1.1])

plt.subplot(122)
plt.grid(True, which='both')
plt.axhline(y=0, color='k')
plt.axvline(x=0, color='k')
plt.plot(XK[:, 0][yk==0], XK[:, 1][yk==0], "bs")
plt.plot(XK[:, 0][yk==1], XK[:, 1][yk==1], "g^")
plt.xlabel(r"$x_2$", fontsize=20)
plt.ylabel(r"$x_3$  ", fontsize=20, rotation=0)
plt.annotate(r'$\phi\left(\mathbf{x}\right)$',
             xy=(XK[3, 0], XK[3, 1]),
             xytext=(0.65, 0.50),
             ha="center",
             arrowprops=dict(facecolor='black', shrink=0.1),
             fontsize=18,
            )
plt.plot([-0.1, 1.1], [0.57, -0.1], "r--", linewidth=3)
plt.axis([-0.1, 1.1, -0.1, 1.1])

plt.subplots_adjust(right=1)
```

```
#save_fig("kernel_method_plot")
plt.show()
```




```
x1_example = X1D[3, 0]
for landmark in (-2, 1):
    k = gaussian_rbf(np.array([[x1_example]]), np.array([[landma
rk]]), gamma)
    print("Phi({}, {}) = {}".format(x1_example, landmark, k))
```

```
Phi(-1.0, -2) = [ 0.74081822]
Phi(-1.0, 1) = [ 0.30119421]
```

Using a Gaussian RBF Kernel

```
rbf_kernel_svm_clf = Pipeline((
        ("scaler", StandardScaler()),
        ("svm_clf", SVC(kernel="rbf", gamma=5, C=0.001))
    ))
rbf_kernel_svm_clf.fit(X, y)

from sklearn.svm import SVC

gamma1, gamma2 = 0.1, 5
C1, C2 = 0.001, 1000
hyperparams = (gamma1, C1), (gamma1, C2), (gamma2, C1), (gamma2,
 C2)
```

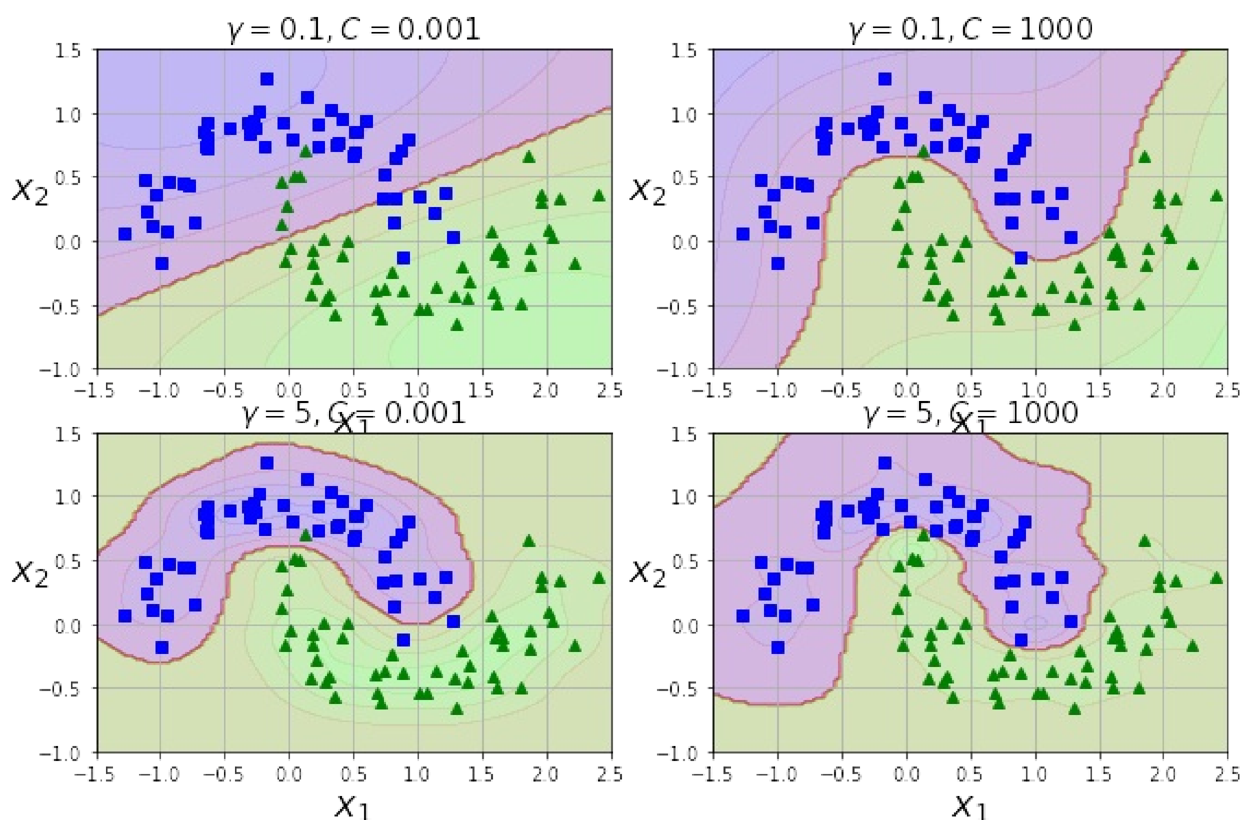
```
svm_clfs = []
for gamma, C in hyperparams:
    rbf_kernel_svm_clf = Pipeline((
            ("scaler", StandardScaler()),
            ("svm_clf", SVC(kernel="rbf", gamma=gamma, C=C))
        ))
    rbf_kernel_svm_clf.fit(X, y)
    svm_clfs.append(rbf_kernel_svm_clf)

plt.figure(figsize=(11, 7))

for i, svm_clf in enumerate(svm_clfs):
    plt.subplot(221 + i)
    plot_predictions(svm_clf, [-1.5, 2.5, -1, 1.5])
    plot_dataset(X, y, [-1.5, 2.5, -1, 1.5])
    gamma, C = hyperparams[i]
    plt.title(r"$\gamma = {}, C = {}$".format(gamma, C), fontsiz
e=16)

#save_fig("moons_rbf_svc_plot")
plt.show()

# below: model trained with different values of gamma and C.
# GAMMA:
# bigger gamma = narrower bell curve, so each instance's area of
 influence = smaller.
# smaller gamma: bigger bell curve = smoother decision boundary.
```



Computational Complexity

- **LinearSVC** class: based on *liblinear* library. Doesn't support kernel trick. Scales linearly to #instances and #features; training complexity ~O(mxn).
- **SVC** class: based on *libsvm* library. Does support kernel trick. Training complexity is O(m^2xn) to O(m^3xn) = MUCH slower on larger training datasets.

SVM Regression (Linear & Non-Linear)

- Objectives: 1) fit max #instances *on* the street; 2) find min #margin violations (instances "off" the street).
- Width controlled by epsilon hyperparameter.
- Below: random linear dataset. two training results with different vals of epsilon.

```
from sklearn.svm import LinearSVR
import numpy.random as rnd

rnd.seed(42)
m = 50
X = 2 * rnd.rand(m, 1)
y = (4 + 3 * X + rnd.randn(m, 1)).ravel()

svm_reg1 = LinearSVR(epsilon=1.5)
svm_reg2 = LinearSVR(epsilon=0.5)
svm_reg1.fit(X, y)
svm_reg2.fit(X, y)

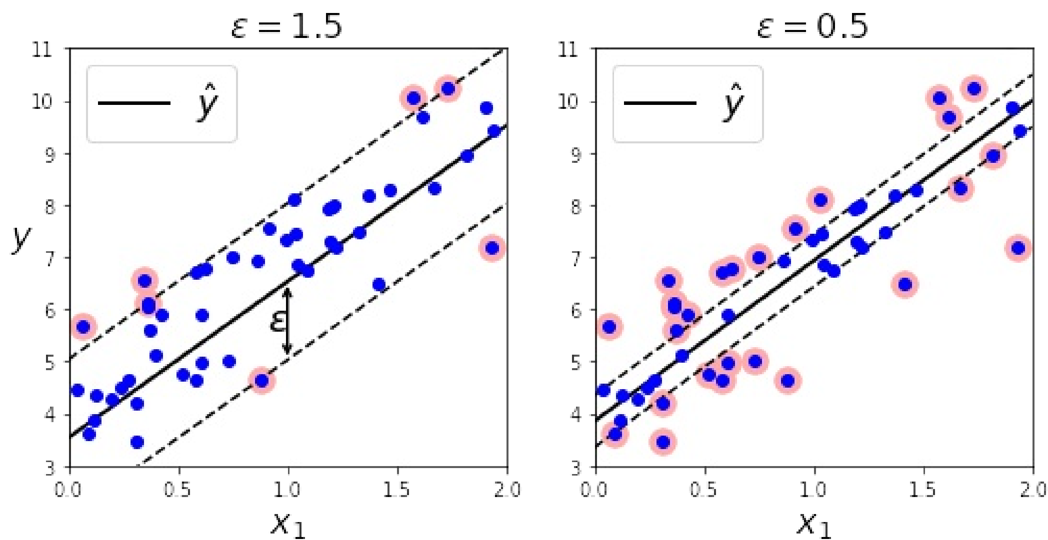
def find_support_vectors(svm_reg, X, y):
    y_pred = svm_reg.predict(X)
    off_margin = (np.abs(y - y_pred) >= svm_reg.epsilon)
    return np.argwhere(off_margin)

svm_reg1.support_ = find_support_vectors(svm_reg1, X, y)
svm_reg2.support_ = find_support_vectors(svm_reg2, X, y)

eps_x1 = 1
eps_y_pred = svm_reg1.predict([[eps_x1]])
```

```
def plot_svm_regression(svm_reg, X, y, axes):
    x1s = np.linspace(axes[0], axes[1], 100).reshape(100, 1)
    y_pred = svm_reg.predict(x1s)
    plt.plot(x1s, y_pred, "k-", linewidth=2, label=r"$\hat{y}$")
    plt.plot(x1s, y_pred + svm_reg.epsilon, "k--")
    plt.plot(x1s, y_pred - svm_reg.epsilon, "k--")
    plt.scatter(X[svm_reg.support_], y[svm_reg.support_], s=180,
 facecolors='#FFAAAA')
    plt.plot(X, y, "bo")
    plt.xlabel(r"$x_1$", fontsize=18)
    plt.legend(loc="upper left", fontsize=18)
    plt.axis(axes)

plt.figure(figsize=(9, 4))
plt.subplot(121)
plot_svm_regression(svm_reg1, X, y, [0, 2, 3, 11])
plt.title(r"$\epsilon = {}$".format(svm_reg1.epsilon), fontsize=
18)
plt.ylabel(r"$y$", fontsize=18, rotation=0)
#plt.plot([eps_x1, eps_x1], [eps_y_pred, eps_y_pred - svm_reg1.e
psilon], "k-", linewidth=2)
plt.annotate(
        '', xy=(eps_x1, eps_y_pred), xycoords='data',
        xytext=(eps_x1, eps_y_pred - svm_reg1.epsilon),
        textcoords='data', arrowprops={'arrowstyle': '<->', 'lin
ewidth': 1.5}
    )
plt.text(0.91, 5.6, r"$\epsilon$", fontsize=20)
plt.subplot(122)
plot_svm_regression(svm_reg2, X, y, [0, 2, 3, 11])
plt.title(r"$\epsilon = {}$".format(svm_reg2.epsilon), fontsize=
18)
#save_fig("svm_regression_plot")
plt.show()
```




```
# Use kernel-ized SVM model to handle nonlinear regression jobs.
```

```
from sklearn.svm import SVR

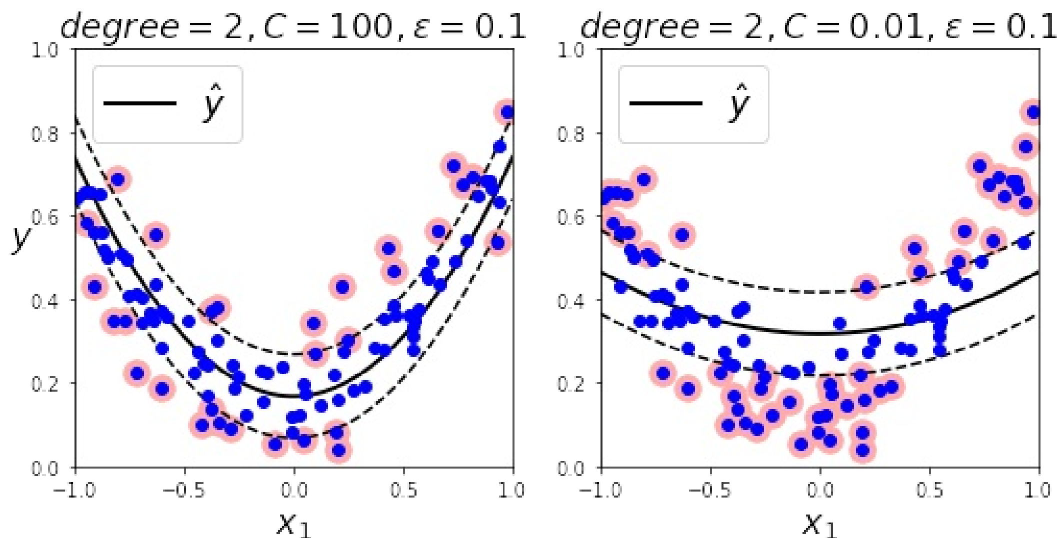
# random quadratic training set.
rnd.seed(42)
m = 100
X = 2 * rnd.rand(m, 1) - 1
y = (0.2 + 0.1 * X + 0.5 * X**2 + rnd.randn(m, 1)/10).ravel()

svm_poly_reg1 = SVR(kernel="poly", degree=2, C=100, epsilon=0.1)
svm_poly_reg2 = SVR(kernel="poly", degree=2, C=0.01, epsilon=0.1
)
svm_poly_reg1.fit(X, y)
svm_poly_reg2.fit(X, y)
```

```
SVR(C=0.01, cache_size=200, coef0=0.0, degree=2, epsilon=0.1, ga
mma='auto',
  kernel='poly', max_iter=-1, shrinking=True, tol=0.001, verbose
=False)
```

```
plt.figure(figsize=(9, 4))
plt.subplot(121)
plot_svm_regression(svm_poly_reg1, X, y, [-1, 1, 0, 1])
plt.title(r"$degree={}, C={}, \epsilon = {}$".format(svm_poly_re
g1.degree, svm_poly_reg1.C, svm_poly_reg1.epsilon), fontsize=18)
plt.ylabel(r"$y$", fontsize=18, rotation=0)
plt.subplot(122)
plot_svm_regression(svm_poly_reg2, X, y, [-1, 1, 0, 1])
plt.title(r"$degree={}, C={}, \epsilon = {}$".format(svm_poly_re
g2.degree, svm_poly_reg2.C, svm_poly_reg2.epsilon), fontsize=18)
#save_fig("svm_with_polynomial_kernel_plot")
plt.show()

# left: little regularization (large C)
# right: much more regularization (little C)
```



Under the Hood

- conventions: b = bias term; w = feature weights vector.

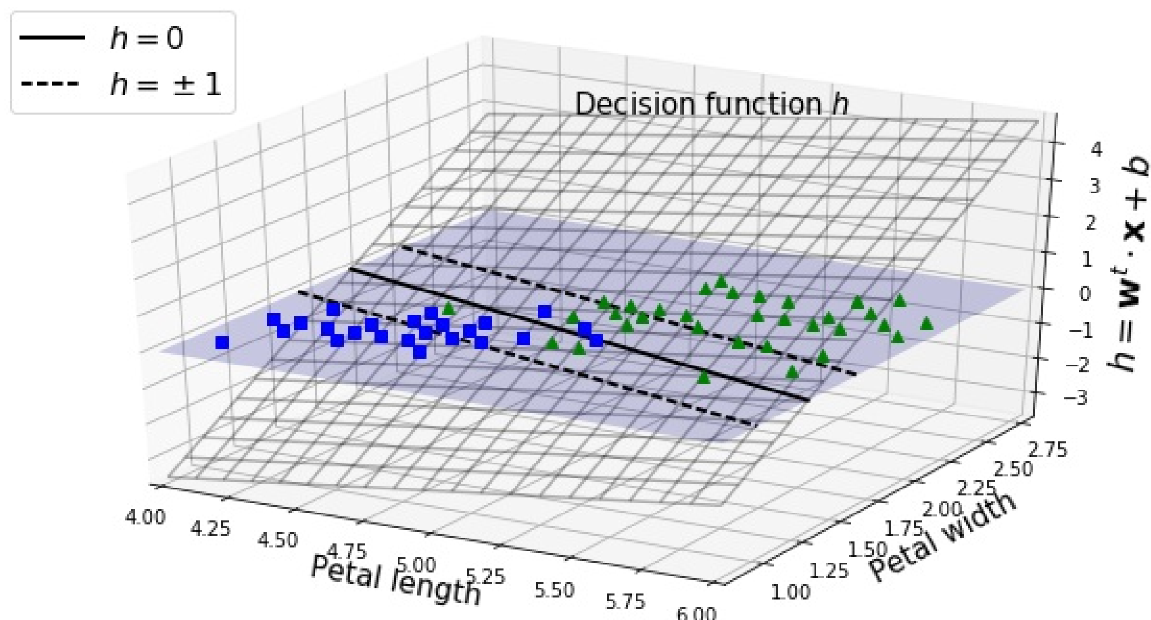
```
iris = datasets.load_iris()
X = iris["data"][:, (2, 3)]  # petal length, petal width
y = (iris["target"] == 2).astype(np.float64)  # Iris-Virginica
```

```
from mpl_toolkits.mplot3d import Axes3D
```

```
def plot_3D_decision_function(ax, w, b, x1_lim=[4, 6], x2_lim=[0
.8, 2.8]):
    x1_in_bounds = (X[:, 0] > x1_lim[0]) & (X[:, 0] < x1_lim[1])
    X_crop = X[x1_in_bounds]
    y_crop = y[x1_in_bounds]
    x1s = np.linspace(x1_lim[0], x1_lim[1], 20)
    x2s = np.linspace(x2_lim[0], x2_lim[1], 20)
    x1, x2 = np.meshgrid(x1s, x2s)
    xs = np.c_[x1.ravel(), x2.ravel()]
    df = (xs.dot(w) + b).reshape(x1.shape)
    m = 1 / np.linalg.norm(w)
    boundary_x2s = -x1s*(w[0]/w[1])-b/w[1]
    margin_x2s_1 = -x1s*(w[0]/w[1])-(b-1)/w[1]
    margin_x2s_2 = -x1s*(w[0]/w[1])-(b+1)/w[1]
    ax.plot_surface(x1s, x2, 0, color="b", alpha=0.2, cstride=100
, rstride=100)
    ax.plot(x1s, boundary_x2s, 0, "k-", linewidth=2, label=r"$h=
0$")
    ax.plot(x1s, margin_x2s_1, 0, "k--", linewidth=2, label=r"$h
=\pm 1$")
    ax.plot(x1s, margin_x2s_2, 0, "k--", linewidth=2)
    ax.plot(X_crop[:, 0][y_crop==1], X_crop[:, 1][y_crop==1], 0,
"g^")
    ax.plot_wireframe(x1, x2, df, alpha=0.3, color="k")
    ax.plot(X_crop[:, 0][y_crop==0], X_crop[:, 1][y_crop==0], 0,
"bs")
    ax.axis(x1_lim + x2_lim)
    ax.text(4.5, 2.5, 3.8, "Decision function $h$", fontsize=15)
    ax.set_xlabel(r"Petal length", fontsize=15)
    ax.set_ylabel(r"Petal width", fontsize=15)
    ax.set_zlabel(r"$h = \mathbf{w}^t \cdot \mathbf{x} + b$", fo
ntsize=18)
    ax.legend(loc="upper left", fontsize=16)

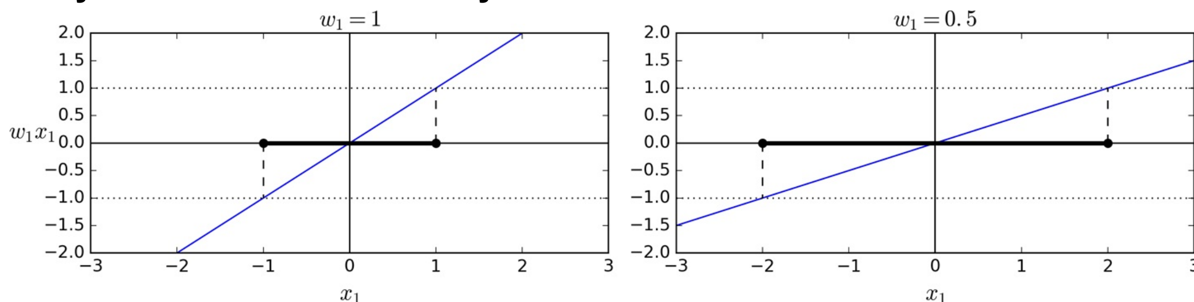
fig = plt.figure(figsize=(11, 6))
ax1 = fig.add_subplot(111, projection='3d')
plot_3D_decision_function(ax1, w=svm_clf2.coef_[0], b=svm_clf2.i
ntercept_[0])
```

```
#save_fig("iris_3D_plot")
plt.show()
```



Training Objectives

- Slope of a decision function equals a weight vector's **norm** (||w||)
- Divide slope by 2 ==> any points where decision function = +1/-1 will be **2x away from decision boundary.**



- So we want minimal ||w|| to get max margins
- If we also want zero margin violations, then decision function needs to be GT1 (positive) and LT1 (negative).
- if soft margins OK - need to define a *slack variable* (C) for tradeoff.

Quadratic programming

- Hard- & soft-margin problems = convex quadratic optimization problems with

linear constraints, ie *quadratic programming* (QP) problems. See Convex Optimization for more info.

todo: The dual problem

todo: Kernelized SVM

todo: Online (incremental learning) SVMs

- Linear SVM classifiers often use **SGD** to find a min-cost solution. SGD converges **much more slowly** than QP-based methods.
- implementation:
- implementation:

Intro

```
import os
import numpy.random as rnd
```

Training & Visualization

```
# load iris dataset & train a DT classifier

from sklearn.datasets import load_iris
from sklearn.tree import DecisionTreeClassifier

iris = load_iris()
X = iris.data[:, 2:] # petal length and width
y = iris.target
tree_clf = DecisionTreeClassifier(max_depth=2)
tree_clf.fit(X, y)
```

```
DecisionTreeClassifier(class_weight=None, criterion='gini', max_
depth=2,
            max_features=None, max_leaf_nodes=None,
            min_impurity_split=1e-07, min_samples_leaf=1,
            min_samples_split=2, min_weight_fraction_leaf=0.0,
            presort=False, random_state=None, splitter='best')
```

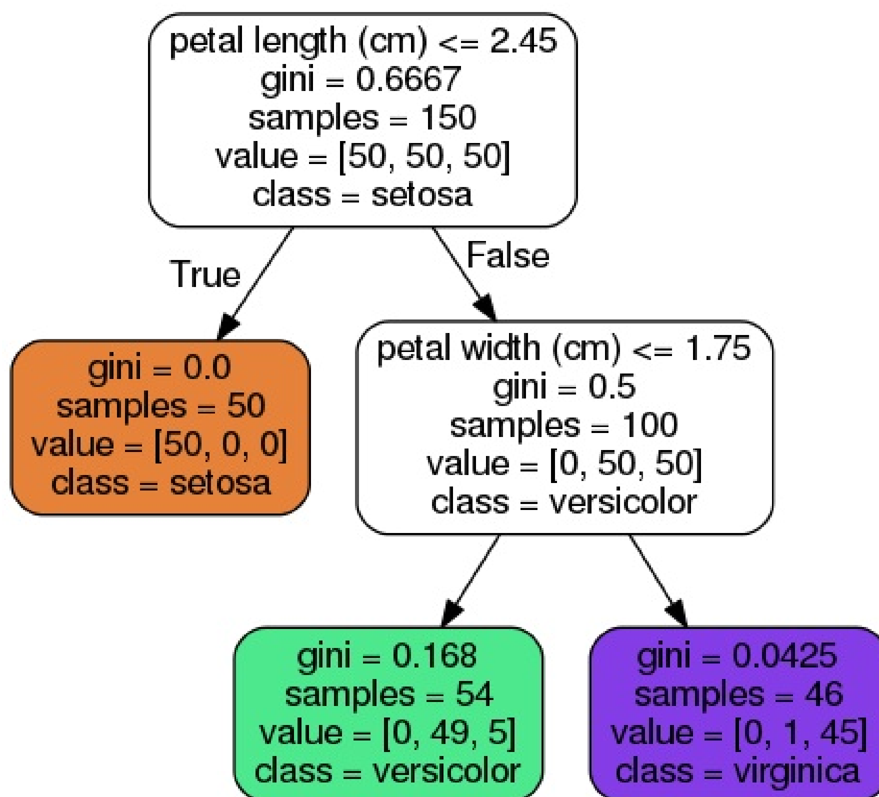
```
# graph it into a .dot file

from sklearn.tree import export_graphviz

def image_path(fig_id):
    #return os.path...
    return fig_id

export_graphviz(
    tree_clf,
    out_file=image_path("iris_tree.dot"),
    feature_names=iris.feature_names[2:],
    class_names=iris.target_names,
    rounded=True,
    filled=True
)
```

```
# convert to PDF or PNG using command-line tool.
! dot -Tpng iris_tree.dot -o iris_tree.png
```



Predictions

- DTs require very little data prep. No feature scaling & centering.
- SciKit uses CART algorithm. (only two children per node.) Other algos, ex ID3, can build DTs with >2 children per node.
- *gini* attribute refers to a node's "impurity" (gini=0 if all applicable training instances belong to same class.)

```
# Plot DT decision boundaries
# Depth=0: root node (petal length=2.45cm)
# Depth=1: right node splits @ 1.75cm
# Stops at max_depth = 2.
# Vertical dotted line shows boundary if max_depth set = 3.

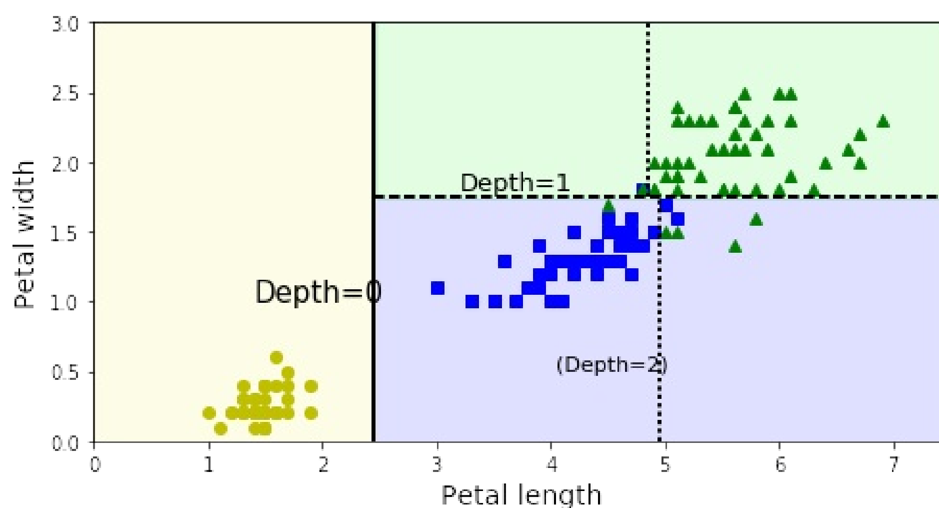
import numpy as np
import matplotlib.pyplot as plt
from matplotlib.colors import ListedColormap

def plot_decision_boundary(clf, X, y, axes=[0, 7.5, 0, 3], iris=True, legend=False, plot_training=True):
    x1s = np.linspace(axes[0], axes[1], 100)
    x2s = np.linspace(axes[2], axes[3], 100)
    x1, x2 = np.meshgrid(x1s, x2s)
    X_new = np.c_[x1.ravel(), x2.ravel()]
    y_pred = clf.predict(X_new).reshape(x1.shape)
    custom_cmap = ListedColormap(['#fafab0','#9898ff','#a0faa0'])
    plt.contourf(x1, x2, y_pred, alpha=0.3, cmap=custom_cmap, linewidth=10)
    if not iris:
        custom_cmap2 = ListedColormap(['#7d7d58','#4c4c7f','#507d50'])
        plt.contour(x1, x2, y_pred, cmap=custom_cmap2, alpha=0.8)
    if plot_training:
        plt.plot(X[:, 0][y==0], X[:, 1][y==0], "yo", label="Iris-Setosa")
        plt.plot(X[:, 0][y==1], X[:, 1][y==1], "bs", label="Iris-Versicolor")
```

```
        plt.plot(X[:, 0][y==2], X[:, 1][y==2], "g^", label="Iris
-Virginica")
        plt.axis(axes)
    if iris:
        plt.xlabel("Petal length", fontsize=14)
        plt.ylabel("Petal width", fontsize=14)
    else:
        plt.xlabel(r"$x_1$", fontsize=18)
        plt.ylabel(r"$x_2$", fontsize=18, rotation=0)
    if legend:
        plt.legend(loc="lower right", fontsize=14)

plt.figure(figsize=(8, 4))
plot_decision_boundary(tree_clf, X, y)
plt.plot([2.45, 2.45], [0, 3], "k-", linewidth=2)
plt.plot([2.45, 7.5], [1.75, 1.75], "k--", linewidth=2)
plt.plot([4.95, 4.95], [0, 1.75], "k:", linewidth=2)
plt.plot([4.85, 4.85], [1.75, 3], "k:", linewidth=2)
plt.text(1.40, 1.0, "Depth=0", fontsize=15)
plt.text(3.2, 1.80, "Depth=1", fontsize=13)
plt.text(4.05, 0.5, "(Depth=2)", fontsize=11)

#save_fig("decision_tree_decision_boundaries_plot")
plt.show()
```



Estimating Class Probabilities

```
# Probability of instance 5cm long, 1.5cm wide belonging to any
one of three nodes above:
print(tree_clf.predict_proba([[5, 1.5]]))

# Return class of highest probability (in this case, class #1.)
print(tree_clf.predict([[5, 1.5]]))
```

```
[[ 0.          0.90740741  0.09259259]]
[1]
```

Training: CART algorithm

- Split training set in two using feature *k* and threshold *t_k*.
- Searches for pair *(k, t_k)* that returns purest subsets, weighted by size.
- Cost function to minimize shown below.

$$J(k, t_k) = \frac{m_{\text{left}}}{m} G_{\text{left}} + \frac{m_{\text{right}}}{m} G_{\text{right}}$$

$$\text{where } \begin{cases} G_{\text{left/right}} \text{ measures the impurity of the left/right subset,} \\ m_{\text{left/right}} \text{ is the number of instances in the left/right subset.} \end{cases}$$

- "Greedy" algorithm; searches for optimum at each level w/o regard for lower levels. Not guaranteed to find optimum solution.

Computational Complexity

- Typical: O(log2(m)) = independent of #features. (So: very fast prediction times.)

Gini Impurity, or Entropy?

- Can use entropy measure by setting *criterion* parameter to "entropy".

$$H_i = - \sum_{\substack{k=1 \\ p_{i,k} \neq 0}}^{n} p_{i,k} \log\left(p_{i,k}\right)$$

- Dataset's entropy = 0 when it contains instances of only one class.

- Can use either; Gini impurity = slightly faster. Entropy tends to build slightly more balanced trees.

Regularization Hyperparameters

- *max_depth* controls max depth of the DT. Reducing *max_depth* regularizes the model, therefore reduces risk of overfit.
- Also: *min_samples_split*, *min_samples_leaf*, *min_weight_fraction_leaf*, *max_leaf_nodes*, *max_features* -- increasing min* _or reducing max_* params will regularize the model.

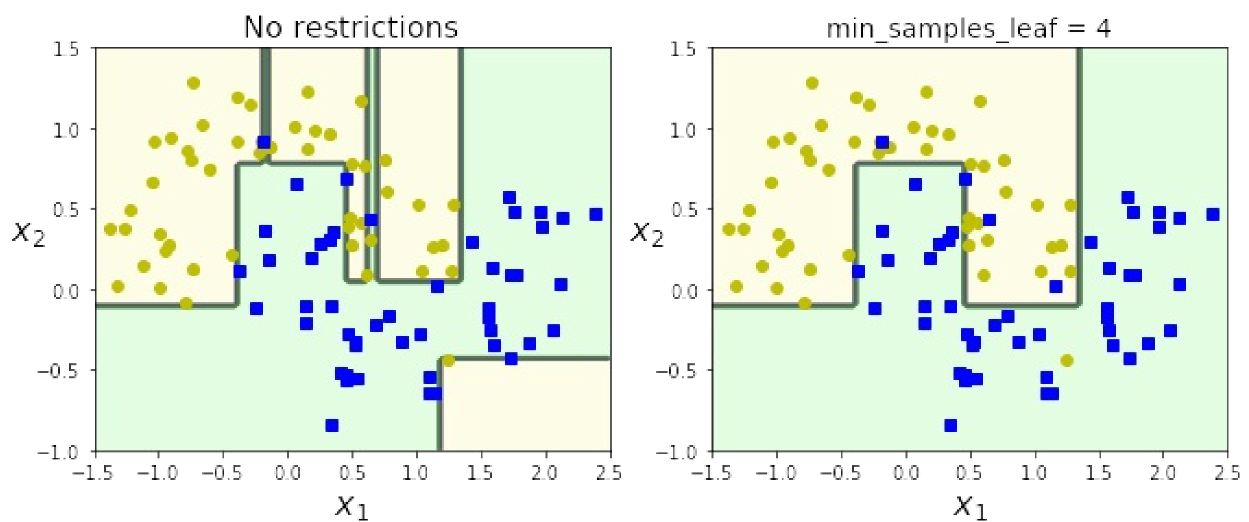
```
# Train two DTs on moons dataset.
# left: default params = no restrictions (case of overfitting)
# right: min_samples_leaf = 4. (better generalization)

from sklearn.datasets import make_moons
Xm, ym = make_moons(n_samples=100, noise=0.25, random_state=53)

deep_tree_clf1 = DecisionTreeClassifier(random_state=42)
deep_tree_clf2 = DecisionTreeClassifier(min_samples_leaf=4, random_state=42)
deep_tree_clf1.fit(Xm, ym)
deep_tree_clf2.fit(Xm, ym)

plt.figure(figsize=(11, 4))
plt.subplot(121)
plot_decision_boundary(deep_tree_clf1, Xm, ym, axes=[-1.5, 2.5, -1, 1.5], iris=False)
plt.title("No restrictions", fontsize=16)
plt.subplot(122)
plot_decision_boundary(deep_tree_clf2, Xm, ym, axes=[-1.5, 2.5, -1, 1.5], iris=False)
plt.title("min_samples_leaf = {}".format(deep_tree_clf2.min_samples_leaf), fontsize=14)

#save_fig("min_samples_leaf_plot")
plt.show()
```



Regression

- Task: Predict a value (instead of a class) for each node.

```
from sklearn.tree import DecisionTreeRegressor

# Quadrat!ic training set + noise
rnd.seed(42)
m = 200
X = rnd.rand(m, 1)
y = 4 * (X - 0.5) ** 2
y = y + rnd.randn(m, 1) / 10

tree_reg1 = DecisionTreeRegressor(random_state=42, max_depth=2)
tree_reg2 = DecisionTreeRegressor(random_state=42, max_depth=3)
tree_reg1.fit(X, y)
tree_reg2.fit(X, y)

def plot_regression_predictions(tree_reg, X, y, axes=[0, 1, -0.2
, 1], ylabel="$y$"):
    x1 = np.linspace(axes[0], axes[1], 500).reshape(-1, 1)
    y_pred = tree_reg.predict(x1)
    plt.axis(axes)
    plt.xlabel("$x_1$", fontsize=18)
    if ylabel:
        plt.ylabel(ylabel, fontsize=18, rotation=0)
    plt.plot(X, y, "b.")
```

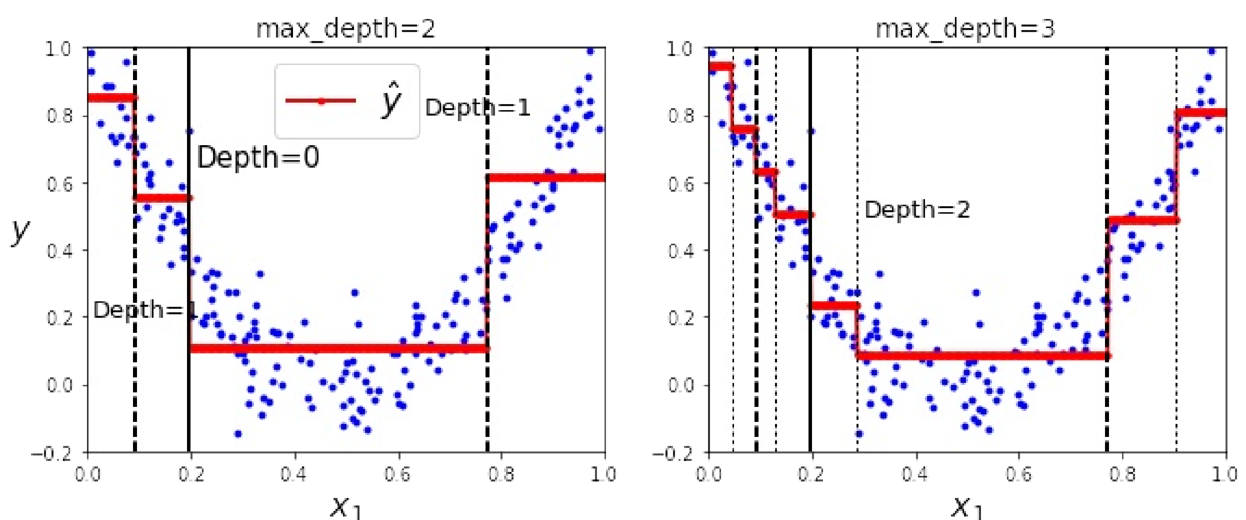
```
    plt.plot(x1, y_pred, "r.-", linewidth=2, label=r"$\hat{y}$")

plt.figure(figsize=(11, 4))
plt.subplot(121)
plot_regression_predictions(tree_reg1, X, y)
for split, style in ((0.1973, "k-"), (0.0917, "k--"), (0.7718, "
k--")):
    plt.plot([split, split], [-0.2, 1], style, linewidth=2)
plt.text(0.21, 0.65, "Depth=0", fontsize=15)
plt.text(0.01, 0.2, "Depth=1", fontsize=13)
plt.text(0.65, 0.8, "Depth=1", fontsize=13)
plt.legend(loc="upper center", fontsize=18)
plt.title("max_depth=2", fontsize=14)

plt.subplot(122)
plot_regression_predictions(tree_reg2, X, y, ylabel=None)
for split, style in ((0.1973, "k-"), (0.0917, "k--"), (0.7718, "
k--")):
    plt.plot([split, split], [-0.2, 1], style, linewidth=2)
for split in (0.0458, 0.1298, 0.2873, 0.9040):
    plt.plot([split, split], [-0.2, 1], "k:", linewidth=1)
plt.text(0.3, 0.5, "Depth=2", fontsize=13)
plt.title("max_depth=3", fontsize=14)

plt.show()

# Predicted value for each region (red line) = avg target value
of instances in that region.
```



- Instead of trying to minimize impurity (classification) DTs now try to minimize MSE:

$$J(k, t_k) = \frac{m_{\text{left}}}{m}\text{MSE}_{\text{left}} + \frac{m_{\text{right}}}{m}\text{MSE}_{\text{right}} \quad \text{where} \begin{cases} \text{MSE}_{\text{node}} = \sum_{i \in \text{node}} \left(\hat{y}_{\text{node}} - y^{(i)}\right)^2 \\ \hat{y}_{\text{node}} = \frac{1}{m_{\text{node}}} \sum_{i \in \text{node}} y^{(i)} \end{cases}$$

```
tree_reg1 = DecisionTreeRegressor(random_state=42)
tree_reg2 = DecisionTreeRegressor(random_state=42, min_samples_l
eaf=10)
tree_reg1.fit(X, y)
tree_reg2.fit(X, y)

x1 = np.linspace(0, 1, 500).reshape(-1, 1)
y_pred1 = tree_reg1.predict(x1)
y_pred2 = tree_reg2.predict(x1)

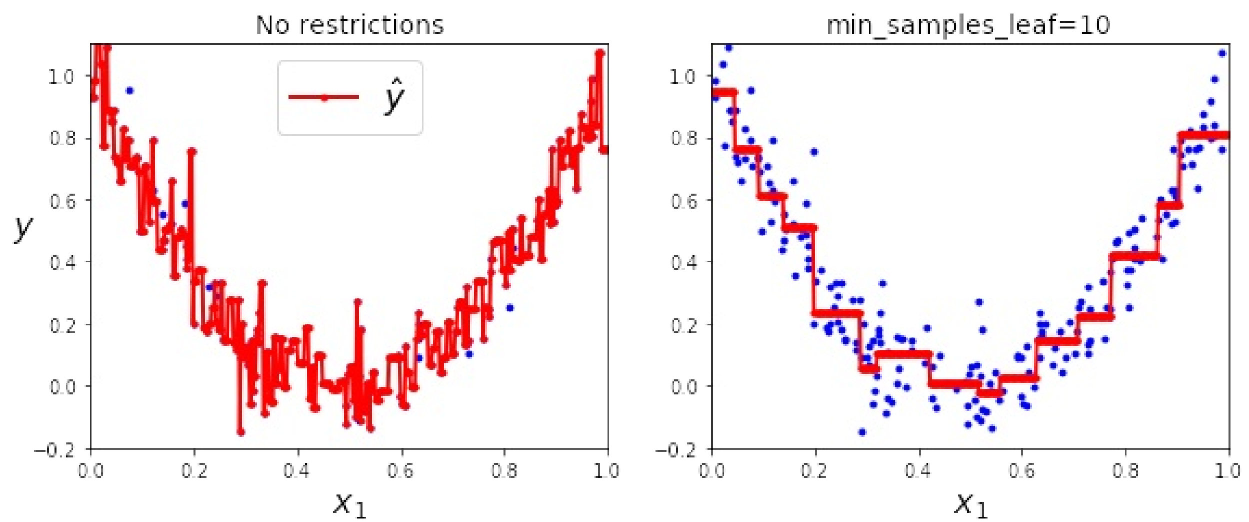
plt.figure(figsize=(11, 4))

plt.subplot(121)
plt.plot(X, y, "b.")
plt.plot(x1, y_pred1, "r.-", linewidth=2, label=r"$\hat{y}$")
plt.axis([0, 1, -0.2, 1.1])
plt.xlabel("$x_1$", fontsize=18)
plt.ylabel("$y$", fontsize=18, rotation=0)
plt.legend(loc="upper center", fontsize=18)
plt.title("No restrictions", fontsize=14)

plt.subplot(122)
plt.plot(X, y, "b.")
plt.plot(x1, y_pred2, "r.-", linewidth=2, label=r"$\hat{y}$")
plt.axis([0, 1, -0.2, 1.1])
plt.xlabel("$x_1$", fontsize=18)
plt.title("min_samples_leaf={}".format(tree_reg2.min_samples_lea
f), fontsize=14)

#save_fig("tree_regression_regularization_plot")
plt.show()

# left: no regularization (default params): overfitting
# right: more reasonable.
```



Instability

- DTs strongly favor orthogonal decision boundaries. They are **sensitive to training set rotations**.
- More generally: DTs are sensitive to training data variations.

```
rnd.seed(6)
Xs = rnd.rand(100, 2) - 0.5
ys = (Xs[:, 0] > 0).astype(np.float32) * 2

angle = np.pi / 4
rotation_matrix = np.array(
    [[np.cos(angle), -np.sin(angle)],
     [np.sin(angle), np.cos(angle)]])

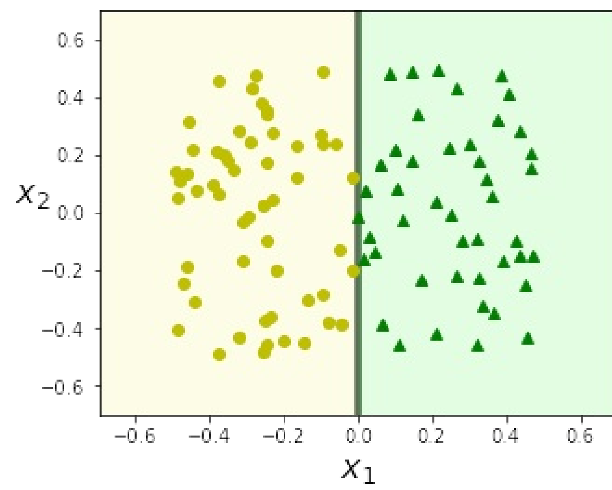
Xsr = Xs.dot(rotation_matrix)

tree_clf_s = DecisionTreeClassifier(random_state=42)
tree_clf_s.fit(Xs, ys)
tree_clf_sr = DecisionTreeClassifier(random_state=42)
tree_clf_sr.fit(Xsr, ys)

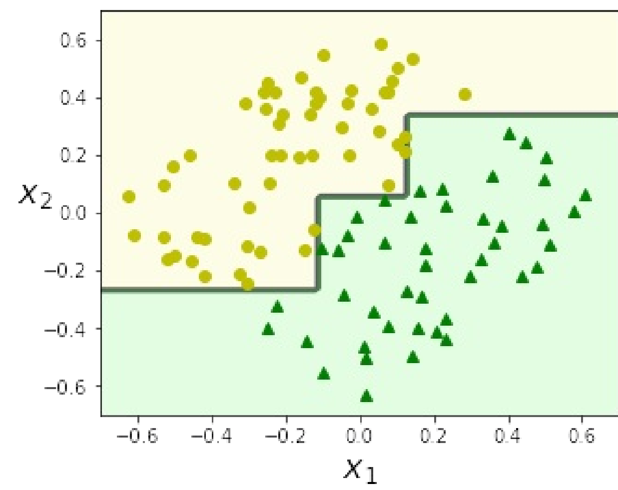
plt.figure(figsize=(11, 4))
plt.subplot(121)
plot_decision_boundary(tree_clf_s, Xs, ys, axes=[-0.7, 0.7, -0.7
, 0.7], iris=False)
plt.subplot(122)
plot_decision_boundary(tree_clf_sr, Xsr, ys, axes=[-0.7, 0.7, -0
.7, 0.7], iris=False)

#save_fig("sensitivity_to_rotation_plot")
plt.show()

# left: std linearly separable dataset
# right: dataset rotated by 45degrees.
```



x_2
x_1
0.6
0.4
0.2
0.0
−0.2
−0.4
−0.6



x_2
x_1
0.6
0.4
0.2
0.0
−0.2
−0.4
−0.6

Intro

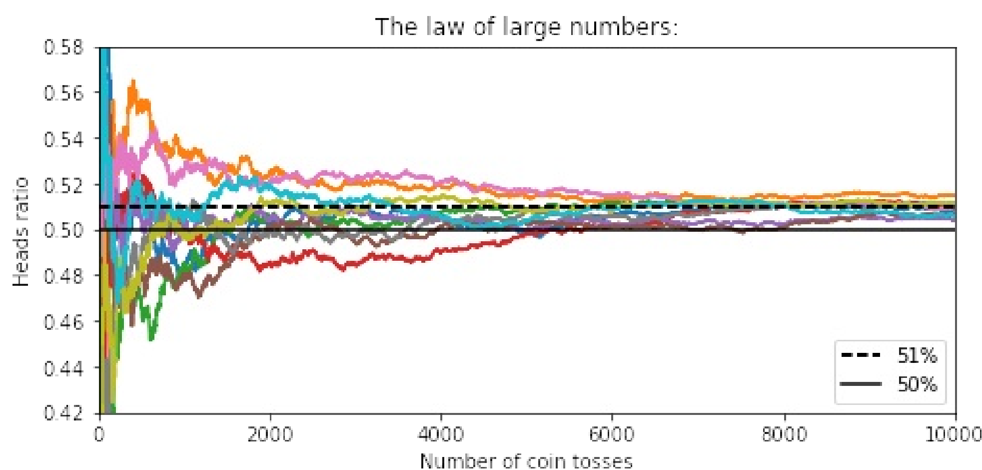
```
import numpy as np
import numpy.random as rnd
import matplotlib.pyplot as plt
```

Voting Classifiers

- Good classifiers can be built by aggregating predictions of various *weaker* classifiers, and returning the class that gets the most votes. (A "hard voting" classifier.)

```
heads_proba = 0.51
coin_tosses = (rnd.rand(10000, 10) < heads_proba).astype(np.int3
2)
cumulative_heads_ratio = np.cumsum(
    coin_tosses, axis=0) / np.arange(1, 10001).reshape(-1, 1)
#cumulative_heads_ratio
```

```
plt.figure(figsize=(8,3.5))
plt.plot(cumulative_heads_ratio)
plt.plot([0, 10000], [0.51, 0.51], "k--", linewidth=2, label="51
%")
plt.plot([0, 10000], [0.5, 0.5], "k-", label="50%")
plt.xlabel("Number of coin tosses")
plt.ylabel("Heads ratio")
plt.legend(loc="lower right")
plt.title("The law of large numbers:")
plt.axis([0, 10000, 0.42, 0.58])
#save_fig("law_of_large_numbers_plot")
plt.show()
```




```
# build a voting classifier in Scikit using three weaker classif
iers

from sklearn.model_selection import train_test_split
from sklearn.datasets import make_moons

# use moons dataset
X, y = make_moons(
    n_samples=500,
    noise=0.30,
    random_state=42)

X_train, X_test, y_train, y_test = train_test_split(
    X, y, random_state=42)

from sklearn.ensemble import RandomForestClassifier
from sklearn.ensemble import VotingClassifier
from sklearn.linear_model import LogisticRegression
from sklearn.svm import SVC

log_clf = LogisticRegression(random_state=42)
rnd_clf = RandomForestClassifier(random_state=42)
svm_clf = SVC(probability=True, random_state=42)

# voting classifier = logistic + random forest + SVC

voting_clf = VotingClassifier(
        estimators=[('lr', log_clf), ('rf', rnd_clf), ('svc', sv
m_clf)],
```

```
        voting='soft'
    )
voting_clf.fit(X_train, y_train)

# let's see how each individual classifier did:

from sklearn.metrics import accuracy_score

for clf in (log_clf, rnd_clf, svm_clf, voting_clf):
    clf.fit(X_train, y_train)
    y_pred = clf.predict(X_test)
    print(clf.__class__.__name__, accuracy_score(y_test, y_pred)
)

# voting classifier did better than 3 individual ones!
```

```
LogisticRegression 0.864
RandomForestClassifier 0.872
SVC 0.888
VotingClassifier 0.912
```

- If all classifiers can estimate class probabilities (they have a predict_proba() method), use Scikit to predict highest class probability, averaged over all individual classifiers. (*soft voting*)

- Often better than hard voting because it gives more weight to highly confident votes. Replace voting="hard" with "soft" & ensure all classifiers can estimate class probabilities. (SVC cannot by default -set probability param to True.)

- This tells SVC to use cross-validation to estimate class probabilities. Slows training times & adds a predict_proba() method).

Bagging & Pasting

- Another approach: use same training algorithm, but apply it to different subsets of the training dataset.
- **bagging**: sampling the dataset **with** replacement.
- **pasting**: sampling the dataset **without** replacement.

- Final prediction = based on an aggregation function.
- Predictions can be made in parallel -- good scaling properties.

```
from sklearn.datasets import make_moons
from sklearn.ensemble import BaggingClassifier
from sklearn.metrics import accuracy_score
from sklearn.tree import DecisionTreeClassifier

# Train ensemble of 500 Decision Tree classifiers
# each using 100 training instances - randomly sampled from trai
ning set
# with replacement.

bag_clf = BaggingClassifier(
        DecisionTreeClassifier(random_state=42),
    n_estimators=500,
    max_samples=100,
    bootstrap=True, # set to False for pasting instead of baggin
g.
    n_jobs=-1,
    random_state=42)

bag_clf.fit(X_train, y_train)
y_pred = bag_clf.predict(X_test)
print(accuracy_score(y_test, y_pred))
```

```
0.904
```

```
tree_clf = DecisionTreeClassifier(random_state=42)
tree_clf.fit(X_train, y_train)
y_pred_tree = tree_clf.predict(X_test)
print(accuracy_score(y_test, y_pred_tree))
```

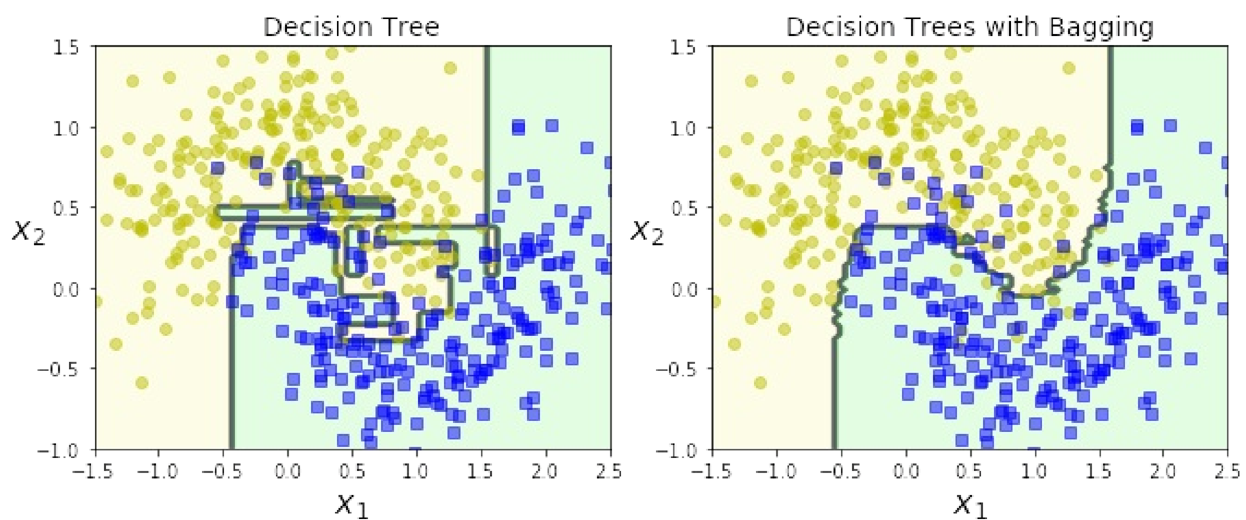
```
0.856
```

```
from matplotlib.colors import ListedColormap

def plot_decision_boundary(clf, X, y, axes=[-1.5, 2.5, -1, 1.5],
 alpha=0.5, contour=True):
    x1s = np.linspace(axes[0], axes[1], 100)
    x2s = np.linspace(axes[2], axes[3], 100)
    x1, x2 = np.meshgrid(x1s, x2s)
    X_new = np.c_[x1.ravel(), x2.ravel()]
    y_pred = clf.predict(X_new).reshape(x1.shape)
    custom_cmap = ListedColormap(['#fafab0','#9898ff','#a0faa0']
)
    plt.contourf(x1, x2, y_pred, alpha=0.3, cmap=custom_cmap, li
newidth=10)
    if contour:
        custom_cmap2 = ListedColormap(['#7d7d58','#4c4c7f','#507
d50'])
        plt.contour(x1, x2, y_pred, cmap=custom_cmap2, alpha=0.8
)
    plt.plot(X[:, 0][y==0], X[:, 1][y==0], "yo", alpha=alpha)
    plt.plot(X[:, 0][y==1], X[:, 1][y==1], "bs", alpha=alpha)
    plt.axis(axes)
    plt.xlabel(r"$x_1$", fontsize=18)
    plt.ylabel(r"$x_2$", fontsize=18, rotation=0)
```

```
plt.figure(figsize=(11,4))
plt.subplot(121)
plot_decision_boundary(tree_clf, X, y)
plt.title("Decision Tree", fontsize=14)

plt.subplot(122)
plot_decision_boundary(bag_clf, X, y)
plt.title("Decision Trees with Bagging", fontsize=14)
#save_fig("decision_tree_without_and_with_bagging_plot")
plt.show()
```



Out of Bag Evaluation

- Bagging: some instances may be sampled multiple times - others not at all. On avg, ~63% of training samples are used. Remainder 37% = "out of bag".
- use *oob_score=True* in Scikit to do automatic oob evaluation after training.

```
# oob_score_: predicts classifier results on test set.
bag_clf = BaggingClassifier(
    DecisionTreeClassifier(),
    n_estimators=500,
    bootstrap=True,
    n_jobs=-1,
    oob_score=True
)
bag_clf.fit(X_train, y_train)
bag_clf.oob_score_
```

```
0.89866666666666661
```

```
# did oob_score_ do a good job?
from sklearn.metrics import accuracy_score
y_pred = bag_clf.predict(X_test)
accuracy_score(y_test,y_pred)
```

```
0.9040000000000003
```

```
# oob decision functionfor each training instance
bag_clf.oob_decision_function_
```

```
array([[ 0.36363636,  0.63636364],
       [ 0.38586957,  0.61413043],
       [ 1.        ,  0.        ],
       [ 0.        ,  1.        ],
       [ 0.        ,  1.        ],
       [ 0.06632653,  0.93367347],
       [ 0.30769231,  0.69230769],
       [ 0.03015075,  0.96984925],
       [ 0.99444444,  0.00555556],
       [ 0.94708995,  0.05291005],
       [ 0.79      ,  0.21      ],
       [ 0.00507614,  0.99492386],
       [ 0.77456647,  0.22543353],
       [ 0.84269663,  0.15730337],
       [ 0.95480226,  0.04519774],
       [ 0.06557377,  0.93442623],
       [ 0.        ,  1.        ],
       [ 0.98      ,  0.02      ],
       [ 0.95505618,  0.04494382],
       [ 1.        ,  0.        ],
       [ 0.01086957,  0.98913043],
       [ 0.3372093 ,  0.6627907 ],
       [ 0.89949749,  0.10050251],
       [ 1.        ,  0.        ],
       [ 0.96666667,  0.03333333],
       [ 0.        ,  1.        ],
       [ 0.99375   ,  0.00625   ],
       [ 1.        ,  0.        ],
       [ 0.        ,  1.        ],
       [ 0.64285714,  0.35714286],
       [ 0.        ,  1.        ],
       [ 1.        ,  0.        ],
```

```
        [ 0.        ,  1.        ],
        [ 0.        ,  1.        ],
        [ 0.14444444,  0.85555556],
        [ 1.        ,  0.        ],
        [ 0.        ,  1.        ],
        [ 0.33152174,  0.66847826],
        [ 0.        ,  1.        ],
        [ 1.        ,  0.        ],
        [ 0.22702703,  0.77297297],
        [ 0.41212121,  0.58787879],
        [ 1.        ,  0.        ],
        [ 1.        ,  0.        ],
        [ 0.        ,  1.        ],
        [ 1.        ,  0.        ],
        [ 1.        ,  0.        ],
        [ 0.02777778,  0.97222222],
        [ 1.        ,  0.        ],
        [ 0.00561798,  0.99438202],
        [ 0.99418605,  0.00581395],
        [ 0.89265537,  0.10734463],
        [ 0.96273292,  0.03726708],
        [ 0.9494382 ,  0.0505618 ],
        [ 0.        ,  1.        ],
        [ 0.03846154,  0.96153846],
        [ 1.        ,  0.        ],
        [ 0.        ,  1.        ],
        [ 0.        ,  1.        ],
        [ 0.00540541,  0.99459459],
        [ 1.        ,  0.        ],
        [ 0.81182796,  0.18817204],
        [ 0.44776119,  0.55223881],
        [ 1.        ,  0.        ],
        [ 0.        ,  1.        ],
        [ 0.63387978,  0.36612022],
        [ 1.        ,  0.        ],
        [ 1.        ,  0.        ],
        [ 0.845     ,  0.155     ],
        [ 1.        ,  0.        ],
        [ 0.55      ,  0.45      ],
        [ 0.13372093,  0.86627907],
```

```
        [ 0.68390805,  0.31609195],
        [ 0.87700535,  0.12299465],
        [ 0.        ,  1.        ],
        [ 0.19487179,  0.80512821],
        [ 0.88324873,  0.11675127],
        [ 1.        ,  0.        ],
        [ 0.        ,  1.        ],
        [ 1.        ,  0.        ],
        [ 0.        ,  1.        ],
        [ 0.07017544,  0.92982456],
        [ 0.0326087 ,  0.9673913 ],
        [ 0.29120879,  0.70879121],
        [ 1.        ,  0.        ],
        [ 0.00487805,  0.99512195],
        [ 0.87700535,  0.12299465],
        [ 0.00543478,  0.99456522],
        [ 0.        ,  1.        ],
        [ 0.        ,  1.        ],
        [ 0.26395939,  0.73604061],
        [ 1.        ,  0.        ],
        [ 0.        ,  1.        ],
        [ 0.        ,  1.        ],
        [ 0.        ,  1.        ],
        [ 0.92227979,  0.07772021],
        [ 0.78494624,  0.21505376],
        [ 0.005     ,  0.995     ],
        [ 1.        ,  0.        ],
        [ 0.1957672 ,  0.8042328 ],
        [ 0.6631016 ,  0.3368984 ],
        [ 0.        ,  1.        ],
        [ 0.03529412,  0.96470588],
        [ 0.4974359 ,  0.5025641 ],
        [ 1.        ,  0.        ],
        [ 0.01785714,  0.98214286],
        [ 0.99465241,  0.00534759],
        [ 0.23626374,  0.76373626],
        [ 0.5270936 ,  0.4729064 ],
        [ 1.        ,  0.        ],
        [ 0.01694915,  0.98305085],
        [ 0.99568966,  0.00431034],
```

```
       [ 0.25988701,  0.74011299],
       [ 0.92982456,  0.07017544],
       [ 1.        ,  0.        ],
       [ 1.        ,  0.        ],
       [ 0.        ,  1.        ],
       [ 0.        ,  1.        ],
       [ 0.80748663,  0.19251337],
       [ 1.        ,  0.        ],
       [ 0.02105263,  0.97894737],
       [ 1.        ,  0.        ],
       [ 1.        ,  0.        ],
       [ 1.        ,  0.        ],
       [ 0.98477157,  0.01522843],
       [ 1.        ,  0.        ],
       [ 0.        ,  1.        ],
       [ 0.92655367,  0.07344633],
       [ 1.        ,  0.        ],
       [ 0.01485149,  0.98514851],
       [ 0.29145729,  0.70854271],
       [ 0.96216216,  0.03783784],
       [ 0.29608939,  0.70391061],
       [ 0.9893617 ,  0.0106383 ],
       [ 0.        ,  1.        ],
       [ 0.        ,  1.        ],
       [ 0.73913043,  0.26086957],
       [ 0.40251572,  0.59748428],
       [ 0.46031746,  0.53968254],
       [ 0.88297872,  0.11702128],
       [ 0.92090395,  0.07909605],
       [ 0.06818182,  0.93181818],
       [ 0.82634731,  0.17365269],
       [ 0.        ,  1.        ],
       [ 0.        ,  1.        ],
       [ 0.01169591,  0.98830409],
       [ 1.        ,  0.        ],
       [ 1.        ,  0.        ],
       [ 1.        ,  0.        ],
       [ 0.00529101,  0.99470899],
       [ 0.        ,  1.        ],
       [ 0.00540541,  0.99459459],
```

```
        [ 0.        ,  1.        ],
        [ 1.        ,  0.        ],
        [ 1.        ,  0.        ],
        [ 0.95238095,  0.04761905],
        [ 1.        ,  0.        ],
        [ 1.        ,  0.        ],
        [ 1.        ,  0.        ],
        [ 0.        ,  1.        ],
        [ 0.34065934,  0.65934066],
        [ 0.23529412,  0.76470588],
        [ 0.00534759,  0.99465241],
        [ 0.00512821,  0.99487179],
        [ 0.31213873,  0.68786127],
        [ 1.        ,  0.        ],
        [ 1.        ,  0.        ],
        [ 0.        ,  1.        ],
        [ 1.        ,  0.        ],
        [ 0.00613497,  0.99386503],
        [ 0.        ,  1.        ],
        [ 1.        ,  0.        ],
        [ 0.        ,  1.        ],
        [ 0.        ,  1.        ],
        [ 1.        ,  0.        ],
        [ 0.00578035,  0.99421965],
        [ 0.63313609,  0.36686391],
        [ 0.9027027 ,  0.0972973 ],
        [ 0.        ,  1.        ],
        [ 0.98963731,  0.01036269],
        [ 1.        ,  0.        ],
        [ 1.        ,  0.        ],
        [ 0.        ,  1.        ],
        [ 0.        ,  1.        ],
        [ 1.        ,  0.        ],
        [ 0.07978723,  0.92021277],
        [ 1.        ,  0.        ],
        [ 0.03645833,  0.96354167],
        [ 0.        ,  1.        ],
        [ 1.        ,  0.        ],
        [ 0.        ,  1.        ],
        [ 0.01818182,  0.98181818],
```

```
       [ 1.        ,  0.        ],
       [ 0.96276596,  0.03723404],
       [ 0.77173913,  0.22826087],
       [ 0.65536723,  0.34463277],
       [ 0.        ,  1.        ],
       [ 0.14832536,  0.85167464],
       [ 1.        ,  0.        ],
       [ 0.96296296,  0.03703704],
       [ 0.97849462,  0.02150538],
       [ 1.        ,  0.        ],
       [ 0.00564972,  0.99435028],
       [ 0.        ,  1.        ],
       [ 0.48924731,  0.51075269],
       [ 0.86243386,  0.13756614],
       [ 0.        ,  1.        ],
       [ 0.        ,  1.        ],
       [ 1.        ,  0.        ],
       [ 0.00543478,  0.99456522],
       [ 0.        ,  1.        ],
       [ 0.96208531,  0.03791469],
       [ 0.        ,  1.        ],
       [ 0.21590909,  0.78409091],
       [ 0.        ,  1.        ],
       [ 1.        ,  0.        ],
       [ 0.        ,  1.        ],
       [ 0.        ,  1.        ],
       [ 0.96875   ,  0.03125   ],
       [ 0.8021978 ,  0.1978022 ],
       [ 1.        ,  0.        ],
       [ 0.00568182,  0.99431818],
       [ 0.05670103,  0.94329897],
       [ 1.        ,  0.        ],
       [ 0.01898734,  0.98101266],
       [ 0.        ,  1.        ],
       [ 0.08888889,  0.91111111],
       [ 1.        ,  0.        ],
       [ 0.77439024,  0.22560976],
       [ 0.        ,  1.        ],
       [ 0.86666667,  0.13333333],
       [ 0.99441341,  0.00558659],
```

```
        [ 0.14634146,  0.85365854],
        [ 0.19487179,  0.80512821],
        [ 1.        ,  0.        ],
        [ 0.        ,  1.        ],
        [ 0.        ,  1.        ],
        [ 0.        ,  1.        ],
        [ 0.26111111,  0.73888889],
        [ 0.96391753,  0.03608247],
        [ 0.        ,  1.        ],
        [ 1.        ,  0.        ],
        [ 1.        ,  0.        ],
        [ 0.        ,  1.        ],
        [ 0.52147239,  0.47852761],
        [ 1.        ,  0.        ],
        [ 0.        ,  1.        ],
        [ 1.        ,  0.        ],
        [ 0.        ,  1.        ],
        [ 0.        ,  1.        ],
        [ 0.07177033,  0.92822967],
        [ 0.13333333,  0.86666667],
        [ 0.99435028,  0.00564972],
        [ 0.01142857,  0.98857143],
        [ 1.        ,  0.        ],
        [ 0.43820225,  0.56179775],
        [ 0.11290323,  0.88709677],
        [ 0.5875    ,  0.4125    ],
        [ 0.60752688,  0.39247312],
        [ 0.        ,  1.        ],
        [ 1.        ,  0.        ],
        [ 0.        ,  1.        ],
        [ 0.        ,  1.        ],
        [ 0.58125   ,  0.41875   ],
        [ 0.        ,  1.        ],
        [ 1.        ,  0.        ],
        [ 0.18032787,  0.81967213],
        [ 0.8150289 ,  0.1849711 ],
        [ 0.07216495,  0.92783505],
        [ 1.        ,  0.        ],
        [ 0.84444444,  0.15555556],
        [ 0.        ,  1.        ],
```

```
       [ 0.        ,  1.        ],
       [ 0.07772021,  0.92227979],
       [ 0.02061856,  0.97938144],
       [ 0.        ,  1.        ],
       [ 0.99473684,  0.00526316],
       [ 0.9076087 ,  0.0923913 ],
       [ 0.15517241,  0.84482759],
       [ 0.96174863,  0.03825137],
       [ 0.00578035,  0.99421965],
       [ 0.60487805,  0.39512195],
       [ 0.03553299,  0.96446701],
       [ 0.98958333,  0.01041667],
       [ 0.82795699,  0.17204301],
       [ 0.        ,  1.        ],
       [ 1.        ,  0.        ],
       [ 0.96174863,  0.03825137],
       [ 0.        ,  1.        ],
       [ 0.        ,  1.        ],
       [ 1.        ,  0.        ],
       [ 0.        ,  1.        ],
       [ 1.        ,  0.        ],
       [ 0.32960894,  0.67039106],
       [ 0.98907104,  0.01092896],
       [ 1.        ,  0.        ],
       [ 0.        ,  1.        ],
       [ 0.        ,  1.        ],
       [ 0.83240223,  0.16759777],
       [ 0.        ,  1.        ],
       [ 1.        ,  0.        ],
       [ 0.79213483,  0.20786517],
       [ 0.94565217,  0.05434783],
       [ 1.        ,  0.        ],
       [ 0.73224044,  0.26775956],
       [ 0.57142857,  0.42857143],
       [ 0.        ,  1.        ],
       [ 0.91666667,  0.08333333],
       [ 0.        ,  1.        ],
       [ 1.        ,  0.        ],
       [ 0.89820359,  0.10179641],
       [ 1.        ,  0.        ],
```

```
         [ 1.        ,  0.        ],
         [ 0.77272727,  0.22727273],
         [ 0.14371257,  0.85628743],
         [ 0.52348993,  0.47651007],
         [ 0.26701571,  0.73298429],
         [ 0.        ,  1.        ],
         [ 0.86486486,  0.13513514],
         [ 0.83536585,  0.16463415],
         [ 0.00591716,  0.99408284],
         [ 1.        ,  0.        ],
         [ 0.99431818,  0.00568182],
         [ 1.        ,  0.        ],
         [ 0.        ,  1.        ],
         [ 0.02659574,  0.97340426],
         [ 0.96067416,  0.03932584],
         [ 0.95767196,  0.04232804],
         [ 1.        ,  0.        ],
         [ 0.47928994,  0.52071006],
         [ 1.        ,  0.        ],
         [ 0.        ,  1.        ],
         [ 0.99459459,  0.00540541],
         [ 0.03157895,  0.96842105],
         [ 1.        ,  0.        ],
         [ 1.        ,  0.        ],
         [ 1.        ,  0.        ],
         [ 0.        ,  1.        ],
         [ 0.96601942,  0.03398058],
         [ 0.        ,  1.        ],
         [ 0.04651163,  0.95348837],
         [ 0.        ,  1.        ],
         [ 0.        ,  1.        ],
         [ 1.        ,  0.        ],
         [ 1.        ,  0.        ],
         [ 0.        ,  1.        ],
         [ 1.        ,  0.        ],
         [ 0.02040816,  0.97959184],
         [ 1.        ,  0.        ],
         [ 0.13917526,  0.86082474],
         [ 0.        ,  1.        ],
         [ 0.01734104,  0.98265896],
```

```
       [ 0.        ,  1.        ],
       [ 0.40952381,  0.59047619],
       [ 0.06818182,  0.93181818],
       [ 0.23195876,  0.76804124],
       [ 1.        ,  0.        ],
       [ 0.98795181,  0.01204819],
       [ 0.20430108,  0.79569892],
       [ 0.99438202,  0.00561798],
       [ 0.        ,  1.        ],
       [ 0.        ,  1.        ],
       [ 1.        ,  0.        ],
       [ 0.97311828,  0.02688172],
       [ 0.31460674,  0.68539326],
       [ 0.98870056,  0.01129944],
       [ 1.        ,  0.        ],
       [ 0.00581395,  0.99418605],
       [ 0.99009901,  0.00990099],
       [ 0.        ,  1.        ],
       [ 0.0304878 ,  0.9695122 ],
       [ 0.97849462,  0.02150538],
       [ 1.        ,  0.        ],
       [ 0.03508772,  0.96491228],
       [ 0.64864865,  0.35135135]])
```

Random Patches - Random Subspaces

- **BaggingClassifier** supports feature sampling. Params: *max_features* and *bootstrap*.
- Very useful when handling high-dimensional datasets.
- "Random patches": sampling features & sampling instances.
- "Random subspaces": sampling features & keeping all instances.

Random Forests

- RF = ensemble of Decision Trees
- Typically trained via bagging
- **RandomForestClassifier**: designed for DT classification
- **RandomForestRegressor**: designed for regression

```
# Train an RF classifier with 500 trees limited to 16 max nodes
each.
# splitter="random": tells RF to search for best feature among
# a random subset of features.

bag_clf = BaggingClassifier(
    DecisionTreeClassifier(
        splitter="random",
        max_leaf_nodes=16,
        random_state=42),

    n_estimators=500,
    max_samples=1.0,
    bootstrap=True,
    n_jobs=-1,
    random_state=42)

bag_clf.fit(X_train, y_train)
y_pred = bag_clf.predict(X_test)
```

```
from sklearn.ensemble import RandomForestClassifier

rnd_clf = RandomForestClassifier(
    n_estimators=500,
    max_leaf_nodes=16,
    n_jobs=-1,
    random_state=42)

rnd_clf.fit(X_train, y_train)
y_pred_rf = rnd_clf.predict(X_test)
```

```
# almost identical predictions
np.sum(y_pred == y_pred_rf) / len(y_pred)
```

```
0.9759999999999998
```

Feature importance

- important features likely to appear closer to root of tree
- unimportant features likely to appear closer to leaves - if at all.
- Scikit finds avg depth of feature appearance across all trees in an RF.

```
# rank features by importance in iris
# #1: petal length: 44%

from sklearn.datasets import load_iris
iris = load_iris()

rnd_clf = RandomForestClassifier(
    n_estimators=500,
    n_jobs=-1,
    random_state=42)

rnd_clf.fit(iris["data"], iris["target"])

for name, importance in zip(
    iris["feature_names"],
    rnd_clf.feature_importances_):
        print(name, "=", importance)
```

```
sepal length (cm) = 0.112492250999
sepal width (cm) = 0.0231192882825
petal length (cm) = 0.441030464364
petal width (cm) = 0.423357996355
```

```
rnd_clf.feature_importances_
```

```
array([ 0.11249225,  0.02311929,  0.44103046,  0.423358  ])
```

```
plt.figure(figsize=(6, 4))

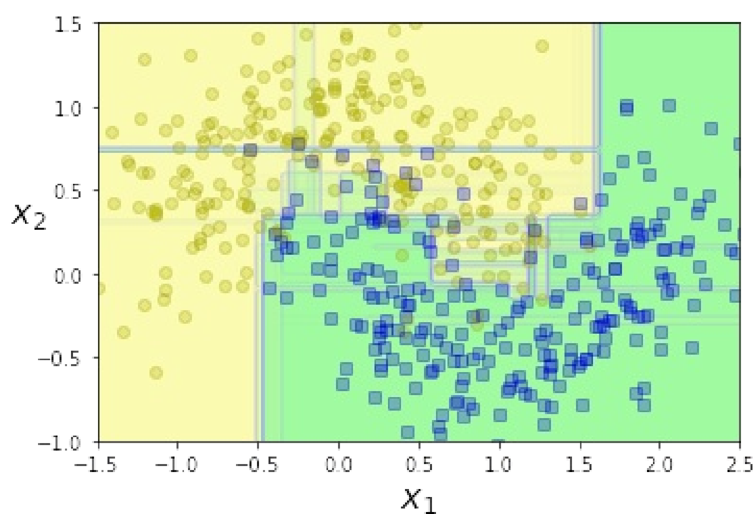
for i in range(15):
    tree_clf = DecisionTreeClassifier(
        max_leaf_nodes=16,
        random_state=42+i)

    indices_with_replacement = rnd.randint(
        0,
        len(X_train),
        len(X_train))

    tree_clf.fit(
        X[indices_with_replacement],
        y[indices_with_replacement])

    plot_decision_boundary(
        tree_clf, X, y,
        axes=[-1.5, 2.5, -1, 1.5],
        alpha=0.02,
        contour=False)

plt.show()
```



Boosting - AdaBoost

- One strategy: pay more attention to training instances that predecessor

underfitted - forces new predictors to concentrate more on the "hard cases".
- **Disadvantage**: results depend on previous classifier (sequential), so algo cannot be parallelized. Not great for scaling.

```
# Plot decision boundaries of five predictors on moons dataset

m = len(X_train)

plt.figure(figsize=(11, 4))
for subplot, learning_rate in ((121, 1), (122, 0.5)):
    sample_weights = np.ones(m)
    for i in range(5):
        plt.subplot(subplot)

        svm_clf = SVC(
            kernel="rbf",
            C=0.05)

        svm_clf.fit(
            X_train, y_train,
            sample_weight=sample_weights)

        y_pred = svm_clf.predict(
            X_train)

        sample_weights[y_pred != y_train] *= (1 + learning_rate)

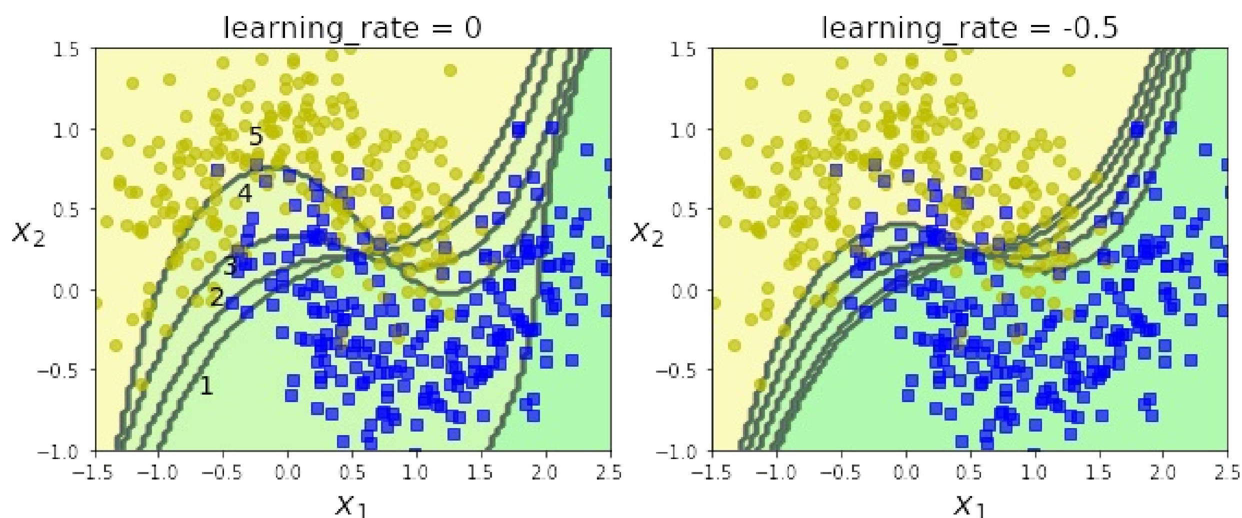
        plot_decision_boundary(
            svm_clf,
            X, y,
            alpha=0.2)

        plt.title("learning_rate = {}".format(learning_rate - 1)
,
                  fontsize=16)

plt.subplot(121)
plt.text(-0.7, -0.65, "1", fontsize=14)
plt.text(-0.6, -0.10, "2", fontsize=14)
plt.text(-0.5,  0.10, "3", fontsize=14)
```

```
plt.text(-0.4,  0.55, "4", fontsize=14)
plt.text(-0.3,  0.90, "5", fontsize=14)
#save_fig("boosting_plot")
plt.show()

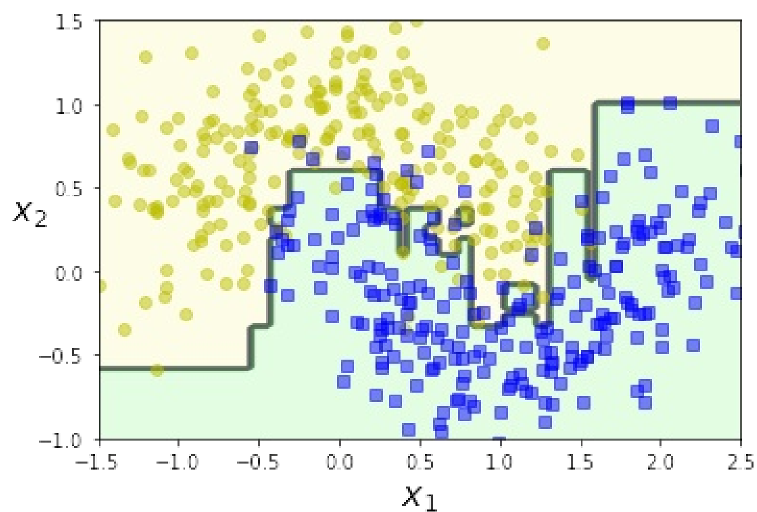
# left: 1st clf gets many wrong, so 2nd clf gets boosted values.
# right: same sequence, but learning rate cut in half.
```




```
# train AdaBoost classifier on 200 decision stumps (DS)
# DS = decision tree with max_depth=1

from sklearn.ensemble import AdaBoostClassifier

ada_clf = AdaBoostClassifier(
        DecisionTreeClassifier(max_depth=1), n_estimators=200,
        algorithm="SAMME.R", learning_rate=0.5, random_state=42
    )
ada_clf.fit(X_train, y_train)
plot_decision_boundary(ada_clf, X, y)
plt.show()
```



Boosting - Gradient Boosting

- Similar to AdaBoost (continually correcting the predecessors in an ensemble. Instead of tweaking instance weights on each iteration, GB fits the predictor to the *residual errors* of the previous predictor.

```
from sklearn.tree import DecisionTreeRegressor

# training set: a noisy quadratic function
rnd.seed(42)
X = rnd.rand(100, 1) - 0.5
y = 3*X[:, 0]**2 + 0.05 * rnd.randn(100)

# train Regressor
tree_reg1 = DecisionTreeRegressor(max_depth=2, random_state=42)
tree_reg1.fit(X, y)

# now train 2nd Regressor using errors made by 1st one.
y2 = y - tree_reg1.predict(X)
tree_reg2 = DecisionTreeRegressor(max_depth=2, random_state=42)
tree_reg2.fit(X, y2)

# now train 3rd Regressor using errors made by 2nd one.
y3 = y2 - tree_reg2.predict(X)
tree_reg3 = DecisionTreeRegressor(max_depth=2, random_state=42)
tree_reg3.fit(X, y3)

X_new = np.array([[0.8]])

# now have ensemble w/ three trees.
y_pred = sum(tree.predict(X_new) for tree in (
    tree_reg1, tree_reg2, tree_reg3))

print(y_pred)
```

```
[ 0.75026781]
```

```
def plot_predictions(
    regressors, X, y, axes,
    label=None,
    style="r-",
    data_style="b.",
    data_label=None):
```

```
    x1 = np.linspace(axes[0], axes[1], 500)

    y_pred = sum(
        regressor.predict(x1.reshape(-1, 1)) for regressor in re
gressors)

    plt.plot(X[:, 0], y, data_style, label=data_label)
    plt.plot(x1, y_pred, style, linewidth=2, label=label)
    if label or data_label:
        plt.legend(loc="upper center", fontsize=16)
    plt.axis(axes)

plt.figure(figsize=(11,11))

plt.subplot(321)
plot_predictions([tree_reg1], X, y, axes=[-0.5, 0.5, -0.1, 0.8],
 label="$h_1(x_1)$", style="g-", data_label="Training set")
plt.ylabel("$y$", fontsize=16, rotation=0)
plt.title("Residuals and tree predictions", fontsize=16)

plt.subplot(322)
plot_predictions([tree_reg1], X, y, axes=[-0.5, 0.5, -0.1, 0.8],
 label="$h(x_1) = h_1(x_1)$", data_label="Training set")
plt.ylabel("$y$", fontsize=16, rotation=0)
plt.title("Ensemble predictions", fontsize=16)

plt.subplot(323)
plot_predictions([tree_reg2], X, y2, axes=[-0.5, 0.5, -0.5, 0.5]
, label="$h_2(x_1)$", style="g-", data_style="k+", data_label="R
esiduals")
plt.ylabel("$y - h_1(x_1)$", fontsize=16)

plt.subplot(324)
plot_predictions([tree_reg1, tree_reg2], X, y, axes=[-0.5, 0.5,
-0.1, 0.8], label="$h(x_1) = h_1(x_1) + h_2(x_1)$")
plt.ylabel("$y$", fontsize=16, rotation=0)

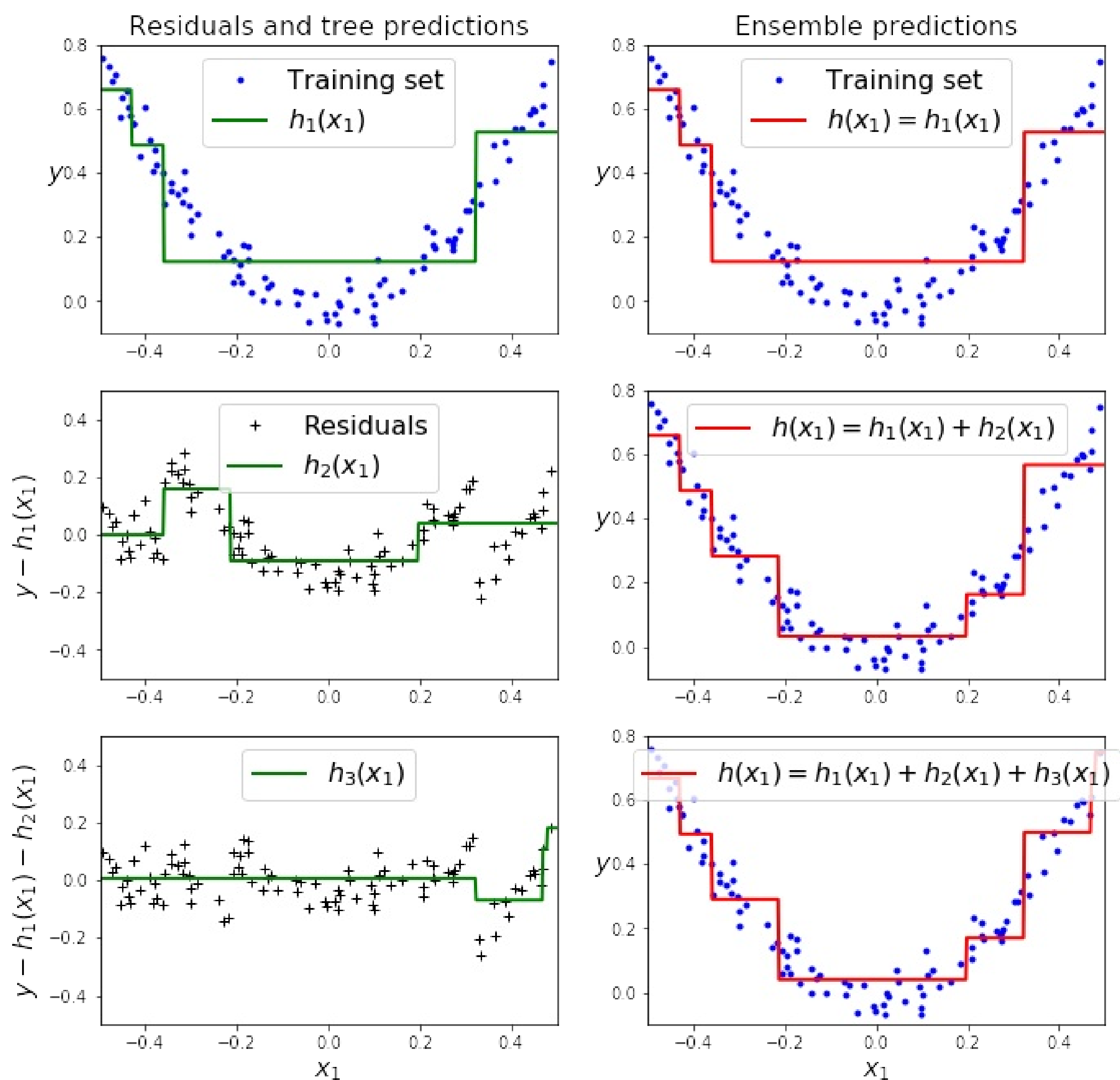
plt.subplot(325)
plot_predictions([tree_reg3], X, y3, axes=[-0.5, 0.5, -0.5, 0.5]
```

```
, label="$h_3(x_1)$", style="g-", data_style="k+")
plt.ylabel("$y - h_1(x_1) - h_2(x_1)$", fontsize=16)
plt.xlabel("$x_1$", fontsize=16)

plt.subplot(326)
plot_predictions([tree_reg1, tree_reg2, tree_reg3], X, y, axes=[
-0.5, 0.5, -0.1, 0.8], label="$h(x_1) = h_1(x_1) + h_2(x_1) + h_
3(x_1)$")
plt.xlabel("$x_1$", fontsize=16)
plt.ylabel("$y$", fontsize=16, rotation=0)

#save_fig("gradient_boosting_plot")
plt.show()

# 1st row: ensemble = only one tree: predictions match 1st tree.
# 2nd row: new tree trained on residual errors of 1st tree.
# 3rd row: "                                              "
# result: ensemble predictions get better as trees are added.
```



- *learning_rate* param controls contribution of each tree. Low values (ex: 0.1) = need more trees in ensemble to fit training set, but predictions usually generalize better. (This is called **shrinkage**.)

```
# two GBRT ensembles trained with low learning rate

from sklearn.ensemble import GradientBoostingRegressor

gbrt = GradientBoostingRegressor(
    max_depth=2,
    n_estimators=3,
    learning_rate=0.1,
    random_state=42)
```

```
gbrt.fit(X, y)

gbrt_slow = GradientBoostingRegressor(
    max_depth=2,
    n_estimators=200,
    learning_rate=0.1,
    random_state=42)

gbrt_slow.fit(X, y)

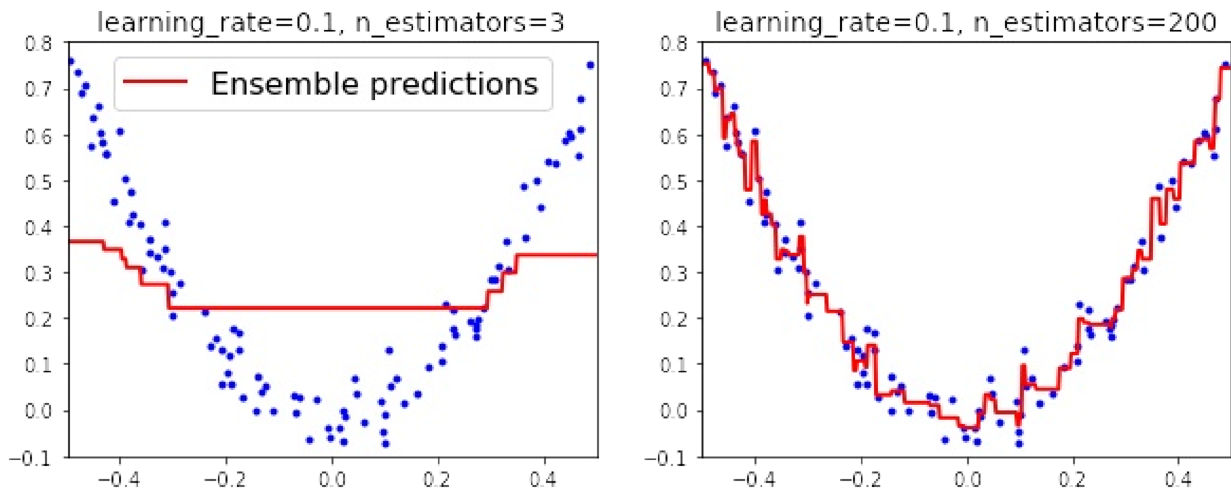
plt.figure(figsize=(11,4))

plt.subplot(121)
plot_predictions(
    [gbrt], X, y,
    axes=[-0.5, 0.5, -0.1, 0.8],
    label="Ensemble predictions")
plt.title("learning_rate={}, n_estimators={}".format(gbrt.learni
ng_rate, gbrt.n_estimators), fontsize=14)

plt.subplot(122)
plot_predictions(
    [gbrt_slow], X, y,
    axes=[-0.5, 0.5, -0.1, 0.8])
plt.title("learning_rate={}, n_estimators={}".format(gbrt_slow.l
earning_rate, gbrt_slow.n_estimators), fontsize=14)

#save_fig("gbrt_learning_rate_plot")
plt.show()

# left: not enough trees (underfits)
# right: too many trees (overfits)
```



- To find optimal number of trees - use early stopping method.
- *staged_predict* method: returns iterator

```
from sklearn.model_selection import train_test_split
from sklearn.metrics import mean_squared_error

X_train, X_val, y_train, y_val = train_test_split(X, y)

# train GRBR regressor with 120 trees

gbrt = GradientBoostingRegressor(
    max_depth=2,
    n_estimators=120,
    learning_rate=0.1,
    random_state=42)

gbrt.fit(X_train, y_train)

# measure MSE validation error at each stage
errors = [mean_squared_error(y_val, y_pred) for y_pred in gbrt.s
taged_predict(X_val)]
errors
```

```
[0.0587714680954524l,
 0.050146609664278821,
 0.042693525239940654,
 0.036758764317358611,
 0.032342621749728441,
```

```
    0.028407668512271105,
    0.024897554253370889,
    0.022344405311247584,
    0.019535997367701449,
    0.017423553892941333,
    0.015298227412102105,
    0.013614891608372095,
    0.0124186540197878 6,
    0.011149507337239 46,
    0.010131360091843384,
    0.0091854704682465919,
    0.0085684302891776056,
    0.0078525358395017328,
    0.0072105819722258777,
    0.0067708705683962693,
    0.0062415649764643415,
    0.0058360573276457243,
    0.0053862983457847987,
    0.0051345071507873903,
    0.0048692096567381805,
    0.0045993749990593299,
    0.0043550054844811968,
    0.0041542481413648245,
    0.0039794595160053785,
    0.0038058301746231277,
    0.0036528925611761264,
    0.0035903310836105469,
    0.0035078898256137104,
    0.0034145667924260869,
    0.0033091498103360911,
    0.0032216349333429491,
    0.0031684358902285465,
    0.0031067035318094903,
    0.0030811367114601672,
    0.0030602631146299077,
    0.003000040093686018,
    0.0029246869254349805,
    0.0028559321605494477,
    0.0028308419421558683,
    0.0028218777360194264,
```

```
  0.0027941065824977074,
  0.0027733228935542496,
  0.0027805517665357811,
  0.0027523772234700978,
  0.0027297064654860348,
  0.0027248578787871292,
  0.0027111390401517179,
  0.0027041926119007326,
  0.0026930464329994377,
  0.0027047076934144398,
  0.0027194180251317295,
  0.0027010027055809748,
  0.0026976053707465464,
  0.0026946405089738347,
  0.0026713744909731395,
  0.0026633491003786457,
  0.0026694977341077202,
  0.0026594592750579836,
  0.0026425819418378605,
  0.0026524409142755744,
  0.0026418897165154491,
  0.0026483360802177103,
  0.0026456393608631189,
  0.0026465080389023671,
  0.0026396693211148074,
  0.002649273120700455,
  0.002643721514468783,
  0.0026463988198929221,
  0.0026333618213948747,
  0.0026314011519099879,
  0.0026349113355268257,
  0.0026387528659342825,
  0.0026345585421650142,
  0.0026355886319374901,
  0.0026310345391991532,
  0.0026519658939712061,
  0.0026467700098620557,
  0.002644982396657l5,
  0.0026475491456891486,
  0.0026474836942911913,
```

```
 0.002653045815536568l,
 0.0026478335004093052,
 0.0026564768881028435,
 0.0026574608795571115,
 0.0026537575609276061,
 0.0026559108292476983,
 0.0026528848367343987,
 0.0026533895549644779,
 0.0026520896622857252,
 0.0026416985817433059,
 0.0026497886163651938,
 0.0026430582537166087,
 0.0026548742317473117,
 0.002660275592603878,
 0.0026582571161537366,
 0.0026570823709750535,
 0.0026557538081706522,
 0.002675470519360824,
 0.0026762761989050578,
 0.0026742086578626454,
 0.0026957941482744232,
 0.0026964801899977998,
 0.0026939578807501376,
 0.0026959742963617757,
 0.0026949319702616616,
 0.0026988916344244736,
 0.0027169473218451121,
 0.0027148017926961689,
 0.0027192710134859655,
 0.0027358435370699618,
 0.0027346474658663323,
 0.0027351047440069571,
 0.0027459941366245631,
 0.0027441324932851491,
 0.002756368378237764]
```

```
# train another GBRT ensemble using optimal #trees

best_n_estimators = np.argmin(errors)
min_error = errors[best_n_estimators]

gbrt_best = GradientBoostingRegressor(
    max_depth=2,
    n_estimators=best_n_estimators,
    learning_rate=0.1,
    random_state=42)

gbrt_best.fit(X_train, y_train)
```

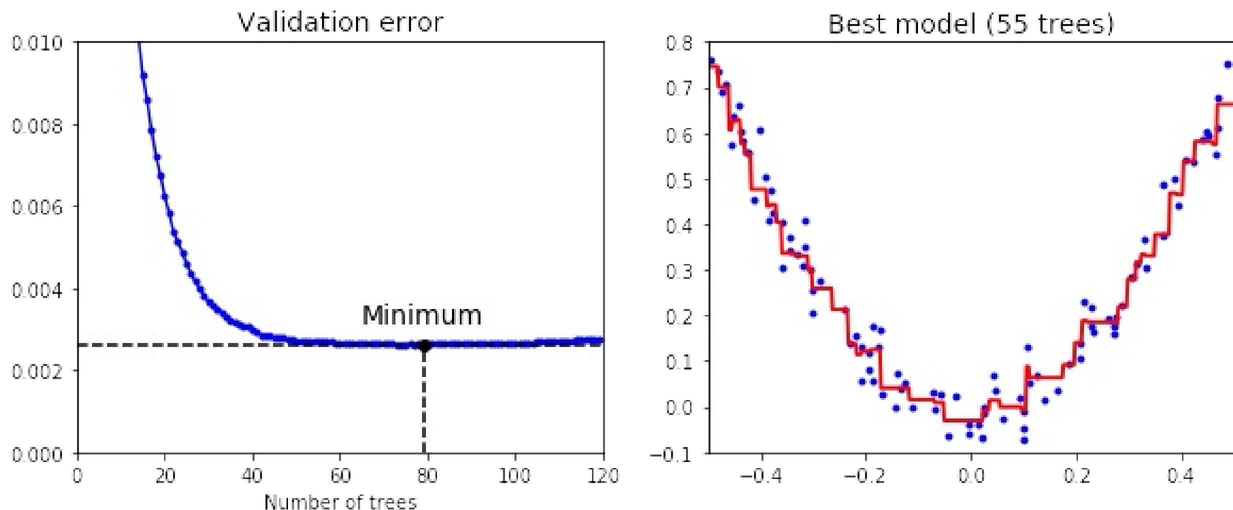
```
GradientBoostingRegressor(alpha=0.9, criterion='friedman_mse', i
nit=None,
             learning_rate=0.1, loss='ls', max_depth=2, max_feat
ures=None,
             max_leaf_nodes=None, min_impurity_split=1e-07,
             min_samples_leaf=1, min_samples_split=2,
             min_weight_fraction_leaf=0.0, n_estimators=79, pres
ort='auto',
             random_state=42, subsample=1.0, verbose=0, warm_sta
rt=False)
```

```
plt.figure(figsize=(11, 4))

plt.subplot(121)
plt.plot(errors, "b.-")
plt.plot([best_n_estimators, best_n_estimators], [0, min_error],
"k--")
plt.plot([0, 120], [min_error, min_error], "k--")
plt.plot(best_n_estimators, min_error, "ko")
plt.text(best_n_estimators, min_error*1.2, "Minimum", ha="center"
, fontsize=14)
plt.axis([0, 120, 0, 0.01])
plt.xlabel("Number of trees")
plt.title("Validation error", fontsize=14)

plt.subplot(122)
plot_predictions([gbrt_best], X, y, axes=[-0.5, 0.5, -0.1, 0.8])
plt.title("Best model (55 trees)", fontsize=14)

#save_fig("early_stopping_gbrt_plot")
plt.show()
```



- Another method: actually stopping training early
- Implement via *warm_start=True* (tells Scikit to keep existing trees when fit() is called - allowing incremental training.)

```
gbrt = GradientBoostingRegressor(
    max_depth=2,
    n_estimators=1,
    learning_rate=0.1,
    random_state=42,
    warm_start=True)

min_val_error = float("inf")
error_going_up = 0

# 120 estimators.
# stop training with validation error doesn't improve for
# five consecutive iterations

for n_estimators in range(1, 120):
    gbrt.n_estimators = n_estimators
    gbrt.fit(X_train, y_train)
    y_pred = gbrt.predict(X_val)
    val_error = mean_squared_error(y_val, y_pred)

    if val_error < min_val_error:
        min_val_error = val_error
        error_going_up = 0
    else:
        error_going_up += 1
        if error_going_up == 5:
            break  # early stopping

print(gbrt.n_estimators)
```

```
59
```

Stacking

- Instead of using a voting function to aggregate an ensemble's predictor outputs, instead train a model to do the aggregation. ("blending".)
- Blender training: common approach = use a *holdout set*.

```
# todo: stacking implementation
```

Intro

- Dimesionality reduction is lossy. It may speed up training but can degrade result quality. Also makes pipelines more complex. Try using original data before considering dimensionality reduction.
- Very useful for visualization (2D, 3D representations more intuitive.)
- Two main approaches: **projection**, **manifold learning**.
- Three most popular techniques: **PCA**, **Kernel PCA**, **LLE**.

Curse of Dimensionality

- Many things behave differently in high-D space.

1) Most points in high-D hypercube will be very close to a border.

2) Distances between random points much greater (very high probability of sparse matrix representation).

- In 2D: ~0.52
- In 3D: ~0.66
- In 1,000,000D: ~408 ~ sqrt(1000000/6)

Approaches: Projection

- Most dataset features are concentrated in a few dimensions - not uniformly across all. Much learnable training can be found in low-D subspace.

```
import numpy as np
import numpy.random as rnd
```

```
# build a 3D dataset

rnd.seed(4)
m = 60
w1, w2 = 0.1, 0.3
noise = 0.1

angles = rnd.rand(m) * 3 * np.pi / 2 - 0.5
X = np.empty((m, 3))
X[:, 0] = np.cos(angles) + np.sin(angles)/2 + noise * rnd.randn(
m) / 2
X[:, 1] = np.sin(angles) * 0.7 + noise * rnd.randn(m) / 2
X[:, 2] = X[:, 0] * w1 + X[:, 1] * w2 + noise * rnd.randn(m)

# mean-normalize the data
X = X - X.mean(axis=0)

# apply PCA to reduce to 2D
from sklearn.decomposition import PCA

pca = PCA(n_components = 2)
X2D = pca.fit_transform(X)

# recover 3D points projected on 2D plane
X2D_inv = pca.inverse_transform(X2D)
```

```
# utility to draw 3D arrows
from matplotlib.patches import FancyArrowPatch
from mpl_toolkits.mplot3d import proj3d

class Arrow3D(FancyArrowPatch):
    def __init__(self, xs, ys, zs, *args, **kwargs):
        FancyArrowPatch.__init__(self, (0,0), (0,0), *args, **kw
args)
        self._verts3d = xs, ys, zs

    def draw(self, renderer):
        xs3d, ys3d, zs3d = self._verts3d
        xs, ys, zs = proj3d.proj_transform(xs3d, ys3d, zs3d, ren
derer.M)
        self.set_positions((xs[0],ys[0]),(xs[1],ys[1]))
        FancyArrowPatch.draw(self, renderer)

# express plane as function of x,y
axes = [-1.8, 1.8, -1.3, 1.3, -1.0, 1.0]

x1s = np.linspace(axes[0], axes[1], 10)
x2s = np.linspace(axes[2], axes[3], 10)
x1, x2 = np.meshgrid(x1s, x2s)

C = pca.components_
R = C.T.dot(C)
z = (R[0, 2] * x1 + R[1, 2] * x2) / (1 - R[2, 2])
```

```
# plot 3D dataset, plane & projections

import matplotlib.pyplot as plt
from mpl_toolkits.mplot3d import Axes3D

fig = plt.figure(figsize=(10, 10))
ax = fig.add_subplot(111, projection='3d')

X3D_above = X[X[:, 2] > X2D_inv[:, 2]]
X3D_below = X[X[:, 2] <= X2D_inv[:, 2]]
```

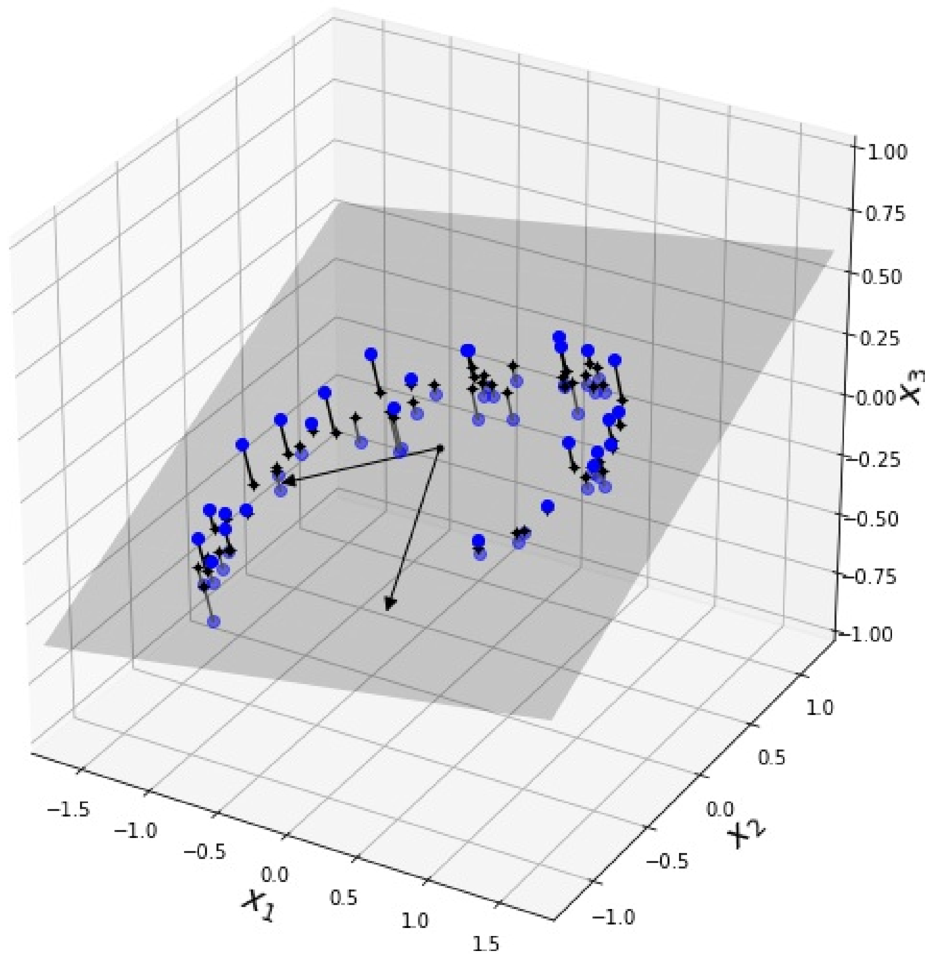
```
ax.plot(X3D_below[:, 0], X3D_below[:, 1], X3D_below[:, 2], "bo",
 alpha=0.5)

ax.plot_surface(x1, x2, z, alpha=0.2, color="k")
np.linalg.norm(C, axis=0)
ax.add_artist(Arrow3D([0, C[0, 0]],[0, C[0, 1]],[0, C[0, 2]], mu
tation_scale=15, lw=1, arrowstyle="-|>", color="k"))
ax.add_artist(Arrow3D([0, C[1, 0]],[0, C[1, 1]],[0, C[1, 2]], mu
tation_scale=15, lw=1, arrowstyle="-|>", color="k"))
ax.plot([0], [0], [0], "k.")

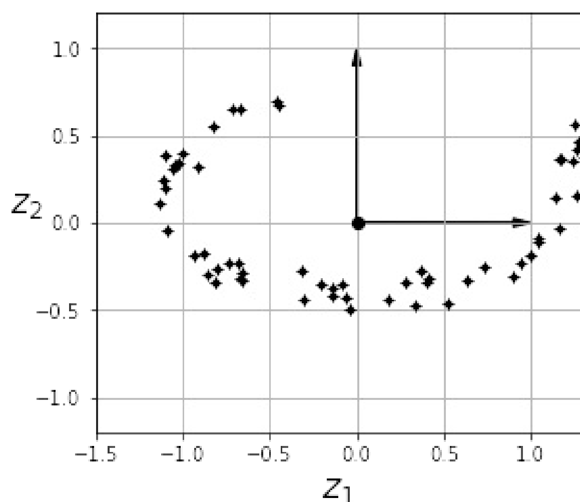
for i in range(m):
    if X[i, 2] > X2D_inv[i, 2]:
        ax.plot([X[i][0], X2D_inv[i][0]], [X[i][1], X2D_inv[i][1
]], [X[i][2], X2D_inv[i][2]], "k-")
    else:
        ax.plot([X[i][0], X2D_inv[i][0]], [X[i][1], X2D_inv[i][1
]], [X[i][2], X2D_inv[i][2]], "k-", color="#505050")

ax.plot(X2D_inv[:, 0], X2D_inv[:, 1], X2D_inv[:, 2], "k+")
ax.plot(X2D_inv[:, 0], X2D_inv[:, 1], X2D_inv[:, 2], "k.")
ax.plot(X3D_above[:, 0], X3D_above[:, 1], X3D_above[:, 2], "bo")
ax.set_xlabel("$x_1$", fontsize=18)
ax.set_ylabel("$x_2$", fontsize=18)
ax.set_zlabel("$x_3$", fontsize=18)
ax.set_xlim(axes[0:2])
ax.set_ylim(axes[2:4])
ax.set_zlim(axes[4:6])

#save_fig("dataset_3d_plot")
plt.show()
```




```
# 2D projection equivalent:
fig = plt.figure()
ax = fig.add_subplot(111, aspect='equal')
ax.plot(X2D[:, 0], X2D[:, 1], "k+")
ax.plot(X2D[:, 0], X2D[:, 1], "k.")
ax.plot([0], [0], "ko")
ax.arrow(0, 0, 0, 1, head_width=0.05, length_includes_head=True,
 head_length=0.1, fc='k', ec='k')
ax.arrow(0, 0, 1, 0, head_width=0.05, length_includes_head=True,
 head_length=0.1, fc='k', ec='k')
ax.set_xlabel("$z_1$", fontsize=18)
ax.set_ylabel("$z_2$", fontsize=18, rotation=0)
ax.axis([-1.5, 1.3, -1.2, 1.2])
ax.grid(True)
plt.show()
```



Approaches: Manifolds

- Manifolds = shapes that can be bent/twisted in higher-D space.
- ex: "Swiss roll" problem

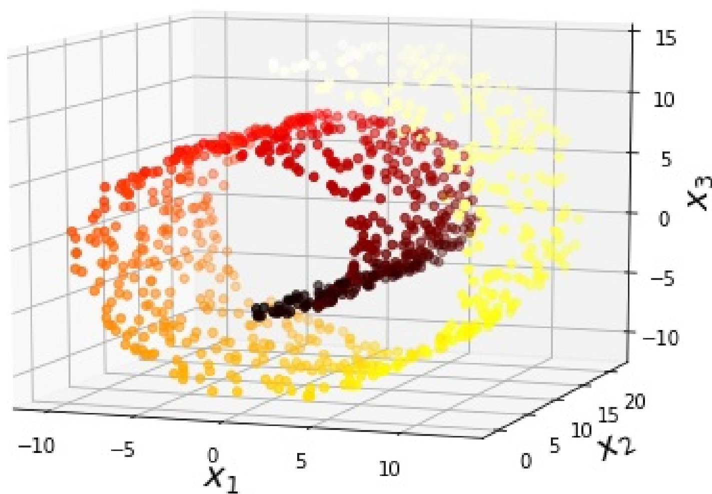
```
# Swiss roll visualization:
from sklearn.datasets import make_swiss_roll
X, t = make_swiss_roll(n_samples=1000, noise=0.2, random_state=42
)

axes = [-11.5, 14, -2, 23, -12, 15]

fig = plt.figure(figsize=(8, 6))
ax = fig.add_subplot(111, projection='3d')

ax.scatter(X[:, 0], X[:, 1], X[:, 2], c=t, cmap=plt.cm.hot)
ax.view_init(10, -70)
ax.set_xlabel("$x_1$", fontsize=18)
ax.set_ylabel("$x_2$", fontsize=18)
ax.set_zlabel("$x_3$", fontsize=18)
ax.set_xlim(axes[0:2])
ax.set_ylim(axes[2:4])
ax.set_zlim(axes[4:6])

#save_fig("swiss_roll_plot")
plt.show()
```

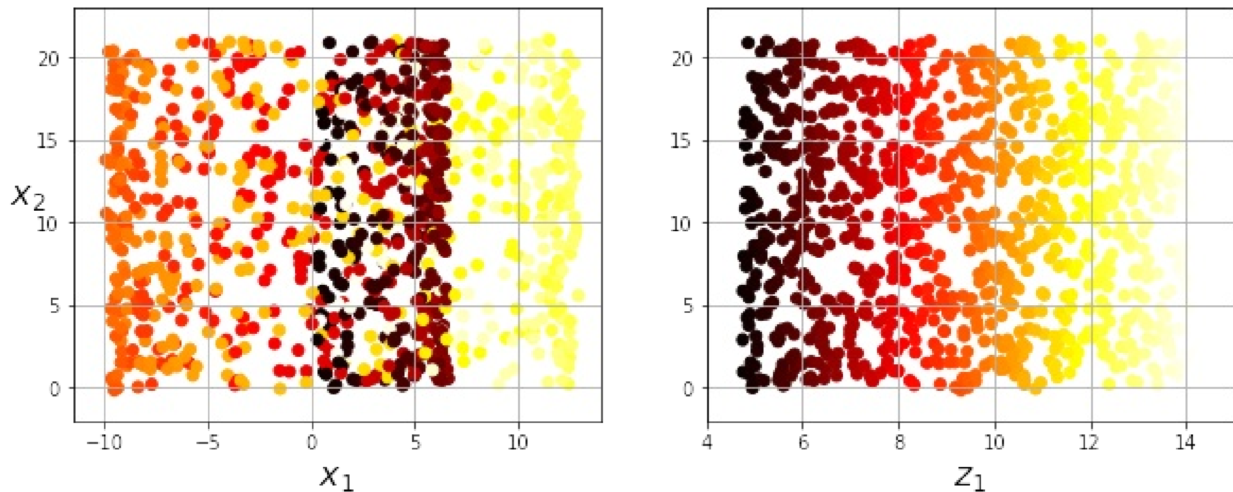



```
# "squashed" swiss roll visualization:
plt.figure(figsize=(11, 4))

plt.subplot(121)
plt.scatter(X[:, 0], X[:, 1], c=t, cmap=plt.cm.hot)
plt.axis(axes[:4])
plt.xlabel("$x_1$", fontsize=18)
plt.ylabel("$x_2$", fontsize=18, rotation=0)
plt.grid(True)

plt.subplot(122)
plt.scatter(t, X[:, 1], c=t, cmap=plt.cm.hot)
plt.axis([4, 15, axes[2], axes[3]])
plt.xlabel("$z_1$", fontsize=18)
plt.grid(True)

#save_fig("squished_swiss_roll_plot")
plt.show()
```




```
from matplotlib import gridspec

axes = [-11.5, 14, -2, 23, -12, 15]

x2s = np.linspace(axes[2], axes[3], 10)
x3s = np.linspace(axes[4], axes[5], 10)
x2, x3 = np.meshgrid(x2s, x3s)

fig = plt.figure(figsize=(6, 5))
ax = plt.subplot(111, projection='3d')

positive_class = X[:, 0] > 5
X_pos = X[positive_class]
X_neg = X[~positive_class]

ax.view_init(10, -70)
ax.plot(X_neg[:, 0], X_neg[:, 1], X_neg[:, 2], "y^")
ax.plot_wireframe(5, x2, x3, alpha=0.5)
ax.plot(X_pos[:, 0], X_pos[:, 1], X_pos[:, 2], "gs")
ax.set_xlabel("$x_1$", fontsize=18)
ax.set_ylabel("$x_2$", fontsize=18)
ax.set_zlabel("$x_3$", fontsize=18)
ax.set_xlim(axes[0:2])
ax.set_ylim(axes[2:4])
ax.set_zlim(axes[4:6])

#save_fig("manifold_decision_boundary_plot1")
plt.show()
```

```
fig = plt.figure(figsize=(5, 4))
ax = plt.subplot(111)

plt.plot(t[positive_class], X[positive_class, 1], "gs")
plt.plot(t[~positive_class], X[~positive_class, 1], "y^")
plt.axis([4, 15, axes[2], axes[3]])
plt.xlabel("$z_1$", fontsize=18)
plt.ylabel("$z_2$", fontsize=18, rotation=0)
plt.grid(True)

#save_fig("manifold_decision_boundary_plot2")
plt.show()

fig = plt.figure(figsize=(6, 5))
ax = plt.subplot(111, projection='3d')

positive_class = 2 * (t[:] - 4) > X[:, 1]
X_pos = X[positive_class]
X_neg = X[~positive_class]
ax.view_init(10, -70)
ax.plot(X_neg[:, 0], X_neg[:, 1], X_neg[:, 2], "y^")
ax.plot(X_pos[:, 0], X_pos[:, 1], X_pos[:, 2], "gs")
ax.set_xlabel("$x_1$", fontsize=18)
ax.set_ylabel("$x_2$", fontsize=18)
ax.set_zlabel("$x_3$", fontsize=18)
ax.set_xlim(axes[0:2])
ax.set_ylim(axes[2:4])
ax.set_zlim(axes[4:6])

#save_fig("manifold_decision_boundary_plot3")
plt.show()

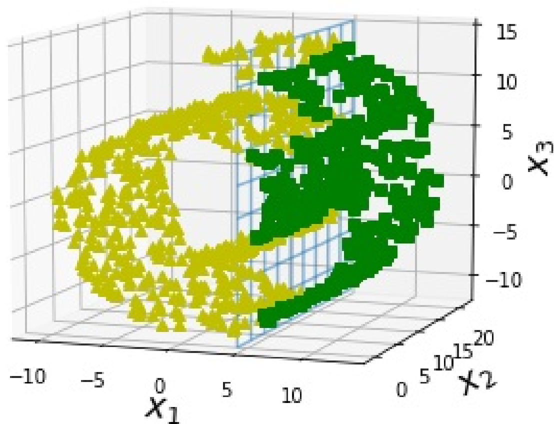
fig = plt.figure(figsize=(5, 4))
ax = plt.subplot(111)

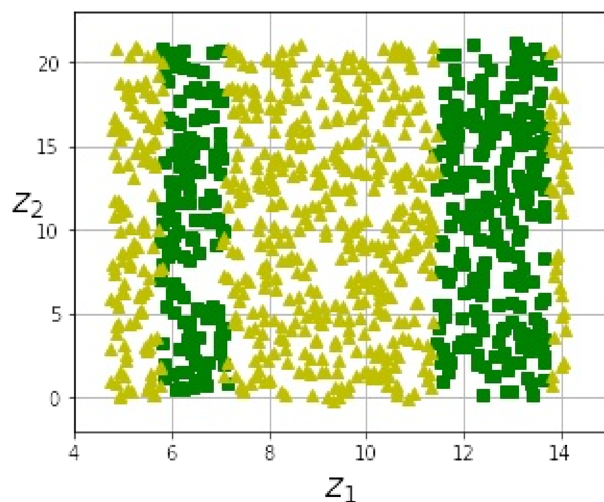
plt.plot(t[positive_class], X[positive_class, 1], "gs")
plt.plot(t[~positive_class], X[~positive_class, 1], "y^")
plt.plot([4, 15], [0, 22], "b-", linewidth=2)
plt.axis([4, 15, axes[2], axes[3]])
```

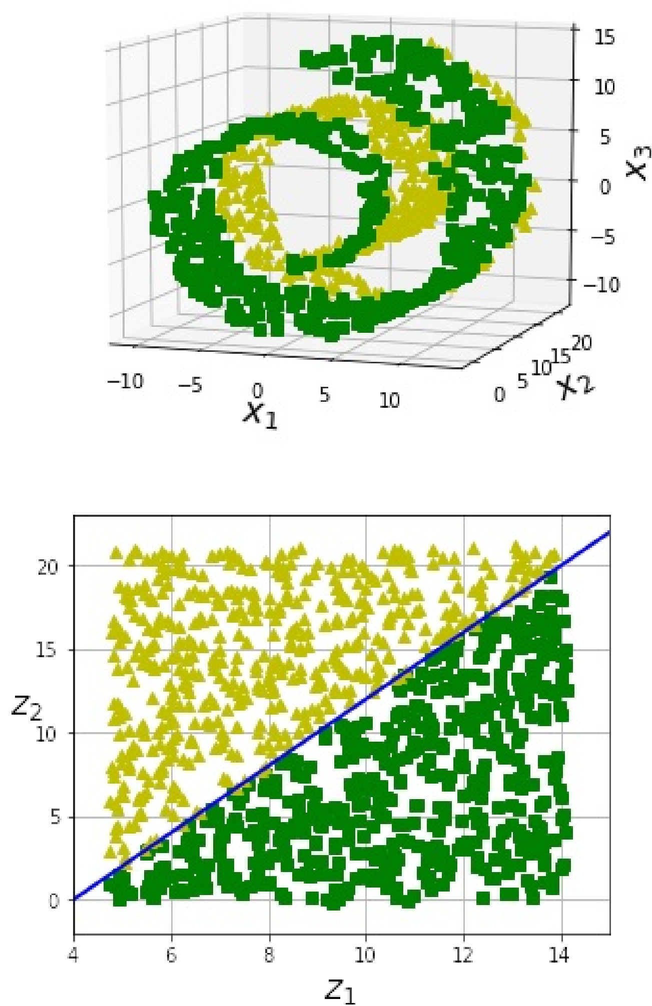
```
plt.xlabel("$z_1$", fontsize=18)
plt.ylabel("$z_2$", fontsize=18, rotation=0)
plt.grid(True)

#save_fig("manifold_decision_boundary_plot4")
plt.show()

# Lesson learned (below):
# Unrolling a dataset to a lower dimension doesn't necessarily l
ead to
# a simpler representation.
```







PCA (Principal Component Analysis)

- Most popular DR algorithm
- 1) Finds hyperplane that lies closest to the data
- 2) Projects data onto it

Preserving Variance

- Below: simple 2D dataset projected onto 3 different axes.
- Projection on solid line preserves the maximum variance. (Therefore less likely to lose information.)

```
angle = np.pi / 5
stretch = 5
```

```
m = 200

rnd.seed(3)
X = rnd.randn(m, 2) / 10
X = X.dot(np.array([[stretch, 0],[0, 1]])) # stretch
X = X.dot([[np.cos(angle), np.sin(angle)], [-np.sin(angle), np.c
os(angle)]]) # rotate

u1 = np.array([np.cos(angle), np.sin(angle)])
u2 = np.array([np.cos(angle - 2 * np.pi/6), np.sin(angle - 2 * n
p.pi/6)])
u3 = np.array([np.cos(angle - np.pi/2), np.sin(angle - np.pi/2)]
)

X_proj1 = X.dot(u1.reshape(-1, 1))
X_proj2 = X.dot(u2.reshape(-1, 1))
X_proj3 = X.dot(u3.reshape(-1, 1))

plt.figure(figsize=(8,4))
plt.subplot2grid((3,2), (0, 0), rowspan=3)
plt.plot([-1.4, 1.4], [-1.4*u1[1]/u1[0], 1.4*u1[1]/u1[0]], "k-",
 linewidth=1)
plt.plot([-1.4, 1.4], [-1.4*u2[1]/u2[0], 1.4*u2[1]/u2[0]], "k--"
, linewidth=1)
plt.plot([-1.4, 1.4], [-1.4*u3[1]/u3[0], 1.4*u3[1]/u3[0]], "k:",
 linewidth=2)
plt.plot(X[:, 0], X[:, 1], "bo", alpha=0.5)
plt.axis([-1.4, 1.4, -1.4, 1.4])
plt.arrow(0, 0, u1[0], u1[1], head_width=0.1, linewidth=5, lengt
h_includes_head=True, head_length=0.1, fc='k', ec='k')
plt.arrow(0, 0, u3[0], u3[1], head_width=0.1, linewidth=5, lengt
h_includes_head=True, head_length=0.1, fc='k', ec='k')
plt.text(u1[0] + 0.1, u1[1] - 0.05, r"$\mathbf{c_1}$", fontsize=
22)
plt.text(u3[0] + 0.1, u3[1], r"$\mathbf{c_2}$", fontsize=22)
plt.xlabel("$x_1$", fontsize=18)
plt.ylabel("$x_2$", fontsize=18, rotation=0)
plt.grid(True)

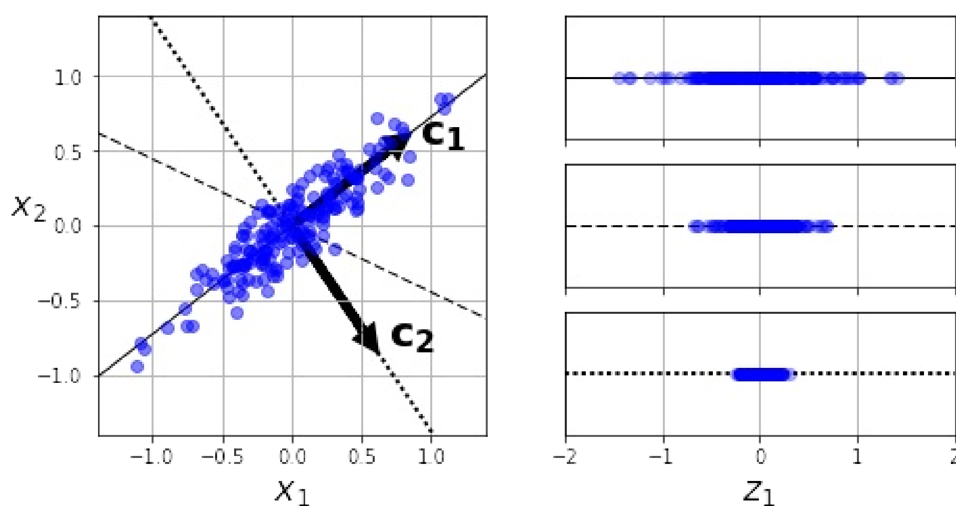
plt.subplot2grid((3,2), (0, 1))
```

```
plt.plot([-2, 2], [0, 0], "k-", linewidth=1)
plt.plot(X_proj1[:, 0], np.zeros(m), "bo", alpha=0.3)
plt.gca().get_yaxis().set_ticks([])
plt.gca().get_xaxis().set_ticklabels([])
plt.axis([-2, 2, -1, 1])
plt.grid(True)

plt.subplot2grid((3,2), (1, 1))
plt.plot([-2, 2], [0, 0], "k--", linewidth=1)
plt.plot(X_proj2[:, 0], np.zeros(m), "bo", alpha=0.3)
plt.gca().get_yaxis().set_ticks([])
plt.gca().get_xaxis().set_ticklabels([])
plt.axis([-2, 2, -1, 1])
plt.grid(True)

plt.subplot2grid((3,2), (2, 1))
plt.plot([-2, 2], [0, 0], "k:", linewidth=2)
plt.plot(X_proj3[:, 0], np.zeros(m), "bo", alpha=0.3)
plt.gca().get_yaxis().set_ticks([])
plt.axis([-2, 2, -1, 1])
plt.xlabel("$z_1$", fontsize=18)
plt.grid(True)

#save_fig("pca_best_projection")
plt.show()
```



Principal Components

- PCA finds axis responsible for largest amount of variance in dataset.
- Also finds 2nd axis, responsible for next largest amount.
- If higher-D dataset, PCA also finds 3rd axis...
- Repeat for # of dimensions in the dataset.
- Each axis vector is called a **principal component**. (PC)

- PCs found using **Singular Value Decomposition (SVD)**, a matrix factorization technique.

- SVD decomposes training set matrix X into dot product of three matrices.
- Note: PCA assumes data is centered around origin. Scikit PCA will adjust data for you if needed.

```
# use NumPy svd() to get principal components of training set,
# then extract 1st two PCs.

X_centered = X - X.mean(axis=0)
U,s,V = np.linalg.svd(X_centered)

c1, c2 = V.T[:,0], V.T[:,1]
print(c1,c2)
```

```
[-0.79644131 -0.60471583] [-0.60471583  0.79644131]
```

Projecting Training Data Down to d Dimensions

- Done by computing dot product of training data (X) by matrix containing the first d principal components (Wd).

```
# project training set onto plane defined by 1st two PCs.

W2 = V.T[:, :2]
X2D = X_centered.dot(W2)
print(X2D)
```

```
[[ -8.96088137e-01   2.61576283e-02]
```

```
 [ -4.53603363e-02  -1.85948860e-01]
 [  1.38359166e-01  -3.11666166e-02]
 [  4.16315780e-02  -6.04371773e-02]
 [  2.18583744e-02  -4.58726693e-02]
 [  6.53868464e-01   1.03673047e-01]
 [ -4.45218566e-01   1.63002740e-01]
 [ -2.52100754e-02  -3.96098381e-02]
 [  2.74828447e-01  -1.47486328e-01]
 [ -4.89804685e-01  -1.19064333e-01]
 [  5.91772943e-01  -6.68825324e-03]
 [ -7.44460369e-01   9.37220434e-03]
 [  5.12230114e-01  -5.91117152e-02]
 [ -3.13266691e-01  -2.12641588e-02]
 [  3.83765553e-01  -1.35145070e-02]
 [ -3.77664930e-01   1.91087392e-01]
 [  6.22192127e-01  -4.81326634e-02]
 [  4.05843018e-01  -2.32002753e-01]
 [  4.62900292e-01  -9.12474313e-02]
 [ -5.62638042e-01  -2.36637544e-02]
 [  8.09046208e-01   8.31463215e-02]
 [  1.80719622e-01  -1.69142171e-01]
 [  2.98447518e-01  -5.11785151e-02]
 [  4.35729072e-01   1.35618235e-02]
 [  1.12339132e+00  -1.68707469e-03]
 [ -5.09329756e-01   7.59538223e-02]
 [ -5.57383436e-01   1.01605408e-01]
 [ -7.42300017e-01  -1.26114063e-01]
 [ -4.19950471e-01  -1.93589947e-01]
 [  3.04360690e-01  -1.83671045e-01]
 [ -5.27822154e-01   1.23668447e-01]
 [  9.38906116e-02   1.80883149e-01]
 [  3.35918853e-01   2.35494590e-02]
 [ -7.52638095e-02  -1.06632109e-01]
 [ -2.24065944e-01   1.90613293e-01]
 [  5.09402619e-01   1.02097673e-01]
 [  7.24520511e-02   1.79929048e-01]
 [ -2.44318685e-01   6.38817933e-02]
 [ -3.23087658e-01   1.94977998e-02]
 [  6.93739438e-01   1.55229402e-01]
 [  6.83626747e-01   3.96804666e-02]
```

```
 [ -3.06255432e-01  -8.88712253e-02]
 [ -7.60306544e-02   1.16609123e-01]
 [  1.28713987e-02  -8.72153282e-02]
 [  1.45854192e+00  -6.50410607e-02]
 [  2.95510472e-01  -4.40211613e-02]
 [  4.78043426e-01   4.92202601e-02]
 [  2.86468892e-01  -3.50652486e-03]
 [ -3.38708502e-01  -9.13086575e-02]
 [  1.44466110e-01   2.20308932e-01]
 [ -4.35369951e-01  -1.37167289e-01]
 [  3.76189231e-02   5.86533049e-02]
 [ -6.17964798e-01   3.26551523e-03]
 [  2.65720436e-01  -6.60652314e-02]
 [ -3.24171713e-01   2.58737230e-02]
 [  2.57613005e-01  -1.20787043e-02]
 [  2.05472262e-01   7.82147166e-02]
 [  3.42825581e-01   5.72823775e-02]
 [ -4.27742475e-01   4.10117883e-02]
 [  4.13138475e-01   1.44642428e-01]
 [  3.37079378e-01   5.11618950e-02]
 [  3.79209530e-01  -1.65052243e-01]
 [ -1.14507596e-01   2.76931986e-02]
 [  3.70774307e-02   2.97937650e-02]
 [  3.24636337e-01  -6.55164554e-02]
 [  7.89476581e-02   1.94034121e-01]
 [ -4.07272297e-01  -5.92029130e-02]
 [ -2.79337894e-01  -5.23406778e-02]
 [  2.25715170e-01   9.21073425e-02]
 [  2.65055409e-01  -1.60611082e-01]
 [  4.51907852e-01   1.97745374e-02]
 [ -6.76603521e-02   1.24160746e-01]
 [  3.55338456e-01   8.20568965e-02]
 [ -2.13673993e-01  -1.80131732e-02]
 [ -4.19919072e-01   4.17639004e-02]
 [  4.27746050e-01   1.17613622e-01]
 [  6.09137236e-01   2.02104565e-02]
 [ -3.16955986e-03   4.38048374e-02]
 [  3.61657526e-01   5.53222800e-03]
 [  6.71393848e-02   1.02545906e-01]
 [  3.96663191e-01   1.70419112e-02]
```

```
 [  1.32276395e-01  -1.25644673e-01]
 [ -1.33866070e+00  -3.39620117e-02]
 [  7.39140839e-01   1.57883864e-01]
 [  5.33339338e-01   4.97439820e-02]
 [ -4.31004868e-01  -7.25504028e-02]
 [  2.15484246e-01  -4.80950244e-02]
 [  6.70354778e-02  -1.59469510e-01]
 [  6.16925576e-01   3.30095454e-04]
 [ -5.21745049e-01   5.35816552e-02]
 [ -8.67032328e-01   5.25304916e-02]
 [  2.54714484e-01  -5.51554532e-03]
 [  1.01594493e+00  -7.32702485e-02]
 [  5.08443285e-01   3.91889488e-02]
 [ -3.23713333e-01  -6.14794166e-02]
 [  2.96394651e-01  -1.47039271e-01]
 [  6.22912229e-02   2.75349476e-02]
 [ -2.22242499e-01  -8.15775375e-02]
 [ -9.97596213e-01   9.98881775e-02]
 [  4.76677524e-02  -5.03263425e-02]
 [  1.59385630e-01   1.98770961e-02]
 [  7.23983658e-03   4.99150260e-02]
 [ -3.88330571e-01   1.29862055e-01]
 [ -5.74324601e-01  -2.17412761e-02]
 [ -1.94317864e-01  -4.14215200e-02]
 [  2.88462890e-01   1.98608595e-01]
 [  2.24621449e-01   2.05254821e-01]
 [  1.72893464e-01   3.03337593e-02]
 [ -5.45686322e-01  -7.71908926e-03]
 [ -1.97062110e-01  -2.69019260e-02]
 [  2.37006918e-01  -1.01833788e-02]
 [  3.20293812e-01   1.71329418e-01]
 [  7.91454892e-02   1.75373382e-01]
 [  1.34116467e+00   3.15311858e-02]
 [ -2.81345950e-01  -3.39245876e-02]
 [ -5.49606098e-01   5.37889594e-02]
 [  1.35299898e-01   4.77632442e-02]
 [ -1.40721018e+00  -3.07936334e-03]
 [ -1.49842206e-01  -4.57709563e-02]
 [ -6.50670013e-02  -1.28928620e-01]
 [  9.28880501e-02   2.49171453e-01]
```

```
 [ -5.35894030e-01   5.42510650e-02]
 [ -5.56858738e-01   1.77869868e-01]
 [ -3.02147552e-01   2.02159209e-01]
 [ -5.36183631e-01   6.74427775e-03]
 [ -8.55623983e-01  -2.52623485e-01]
 [  2.83998192e-01   3.39738443e-02]
 [  4.55220905e-01  -4.60837740e-03]
 [  3.19652268e-01  -5.73147775e-02]
 [ -1.35803712e+00   3.55141032e-02]
 [  2.31421394e-02   1.33432806e-01]
 [  1.15723020e-01   9.23933561e-02]
 [ -1.79851419e-01   1.60298502e-01]
 [  7.07110703e-01   9.29033427e-02]
 [ -5.97227204e-02  -1.15625175e-01]
 [ -5.31804111e-01  -9.70759692e-02]
 [ -8.11573484e-01   3.56003677e-02]
 [ -3.48886570e-01   2.08692440e-03]
 [ -4.49281442e-01  -1.94684913e-01]
 [ -3.70829130e-02  -6.22225918e-02]
 [ -1.42251513e-01  -6.31866218e-03]
 [ -3.07506144e-01  -8.88695185e-02]
 [ -9.25307571e-01   2.13517965e-01]
 [  1.03097886e-01   2.07573310e-03]
 [ -1.47464910e-01   1.24724802e-01]
 [ -9.46748876e-01  -1.39178787e-01]
 [  3.08673618e-01  -9.83295182e-02]
 [  5.58671697e-01  -1.50783004e-01]
 [ -1.79969356e-01  -1.18326957e-01]
 [ -7.54431603e-01   7.63370593e-02]
 [  5.15860621e-01  -9.07637328e-02]
 [  3.02119496e-01   1.47799162e-01]
 [  4.26806702e-01  -1.38986614e-01]
 [  2.28667680e-02   3.92283202e-02]
 [  3.24088211e-01   1.70213975e-01]
 [ -1.22209299e-01   4.41346966e-02]
 [  4.26702773e-01   6.42551604e-03]
 [  8.19816829e-02  -2.24146989e-01]
 [ -1.41063811e-01  -1.81097498e-01]
 [  1.64457703e-02  -1.45681492e-01]
 [ -1.01134103e+00   1.26993172e-02]
```

```
 [ -4.33573355e-01   8.41743009e-02]
 [  6.65856885e-01  -2.12785979e-01]
 [  1.61808196e-01   9.45467292e-02]
 [ -4.82157648e-02   7.89423282e-02]
 [ -4.66117068e-01  -1.57706328e-01]
 [ -1.49009196e-01   1.94946740e-01]
 [  2.88429562e-01  -1.95469647e-01]
 [  1.15854187e-01  -4.26347471e-02]
 [ -2.08314238e-01  -9.22559093e-02]
 [ -4.17242418e-02  -2.21272525e-01]
 [ -4.31285498e-01   1.51023258e-01]
 [ -6.46651297e-01  -1.63796812e-01]
 [ -1.54321959e-01  -3.00319930e-01]
 [ -1.42477982e-01  -8.04414843e-03]
 [  4.96009127e-01   4.62484050e-02]
 [ -7.20255309e-02   9.38505820e-02]
 [ -6.51879918e-02   2.35315556e-02]
 [  9.73713335e-01  -9.40602754e-02]
 [  2.36218522e-01   3.60724373e-02]
 [ -1.06876746e-03   1.51498555e-01]
 [  4.34435954e-01  -1.34060334e-02]
 [ -1.28076019e-01  -2.44578006e-02]
 [ -2.47584073e-01  -6.08927420e-02]
 [ -7.10391471e-01  -4.55521748e-02]
 [ -5.58836834e-01  -1.34202263e-02]
 [ -7.43699081e-01  -1.54131331e-01]
 [ -2.58481292e-01  -4.73208896e-02]
 [ -1.19242387e-01   1.18190473e-01]
 [  5.71157514e-01  -1.18785924e-01]
 [  4.97483707e-01   3.73368260e-02]
 [  9.41952705e-01   3.09293927e-02]
 [  6.44331385e-01  -9.94076651e-02]
 [  1.82692942e-01   4.33205574e-02]
 [  3.13123024e-01  -4.12117592e-02]
 [  2.04403918e-02  -2.54497816e-02]
 [  1.33781786e+00  -1.34015280e-02]
 [ -4.58399199e-02   1.10234115e-01]
 [ -1.02539769e+00   4.64829485e-02]
 [ -3.85549579e-02  -9.59123942e-02]]
```

Scikit PCA

- Uses SVD decomposition as before.
- You can access each PC using *components_* variable. (

```
from sklearn.decomposition import PCA
pca = PCA(n_components = 2)
X2D = pca.fit_transform(X)

print(pca.components_[0])
print(pca.components_.T[:,0])
```

```
[-0.79644131 -0.60471583]
[-0.79644131 -0.60471583]
```

Explained Variance Ratio

- Very useful metric: proportion of dataset's variance along the axis of each PC component.

```
# 95% of dataset variance explained by 1st axis.
print(pca.explained_variance_ratio_)
```

```
[ 0.95369864  0.04630136]
```

Choosing Right #Dimensions

- No need to choose arbitrary #dimensions. Instead pick d that cumulatively accounts for a sufficient amount, ex: 95%.

```
# find minimum d to preserve 95% of training set variance
pca = PCA()
pca.fit(X)
cumsum = np.cumsum(pca.explained_variance_ratio_)
d = np.argmax(cumsum >= 0.95) + 1
print(d)
```

```
1
```

PCA for Compression

- Example applying PCA to MNIST dataset with 95% preservation = results in ~150 features (original = 28x28 = 784)

```
#MNIST compression:
from sklearn.model_selection import train_test_split
from sklearn.datasets import fetch_mldata

#mnist = fetch_mldata('MNIST original')
mnist_path = "./mnist-original.mat"

from scipy.io import loadmat
mnist_raw = loadmat(mnist_path)
mnist = {
    "data": mnist_raw["data"].T,
    "target": mnist_raw["label"][0],
    "COL_NAMES": ["label", "data"],
    "DESCR": "mldata.org dataset: mnist-original",
    }

X, y = mnist["data"], mnist["target"]
X_train, X_test, y_train, y_test = train_test_split(X, y)

X = X_train

pca = PCA()
pca.fit(X)
d = np.argmax(np.cumsum(pca.explained_variance_ratio_) >= 0.95)
+ 1
d
```

```
154
```

```
pca = PCA(n_components=0.95)
X_reduced = pca.fit_transform(X)
pca.n_components_
```

```
154
```

```
# did you hit your 95% minimum?
np.sum(pca.explained_variance_ratio_)
```

```
0.9503623084769206
```

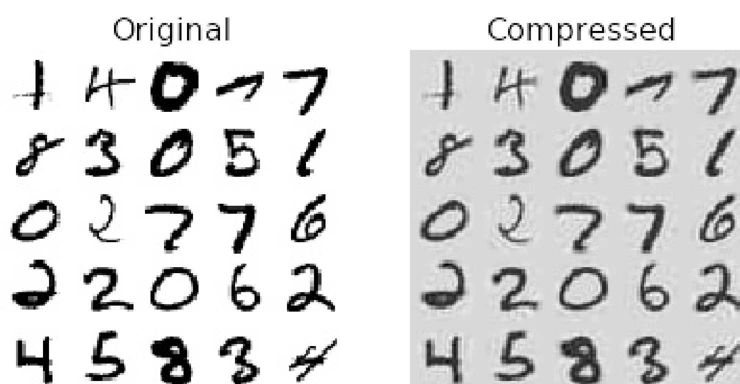
```
# use inverse_transform to decompress back to 784 dimensions
X_mnist = X_train

pca = PCA(n_components = 154)
X_mnist_reduced = pca.fit_transform(X_mnist)
X_mnist_recovered = pca.inverse_transform(X_mnist_reduced)
```

```
import matplotlib
import matplotlib.pyplot as plt

def plot_digits(instances, images_per_row=5, **options):
    size = 28
    images_per_row = min(len(instances), images_per_row)
    images = [instance.reshape(size,size) for instance in instan
ces]
    n_rows = (len(instances) - 1) // images_per_row + 1
    row_images = []
    n_empty = n_rows * images_per_row - len(instances)
    images.append(np.zeros((size, size * n_empty)))
    for row in range(n_rows):
        rimages = images[row * images_per_row : (row + 1) * imag
es_per_row]
        row_images.append(np.concatenate(rimages, axis=1))
    image = np.concatenate(row_images, axis=0)
    plt.imshow(image, cmap = matplotlib.cm.binary, **options)
    plt.axis("off")

plt.figure(figsize=(7, 4))
plt.subplot(121)
plot_digits(X_mnist[::2100])
plt.title("Original", fontsize=16)
plt.subplot(122)
plot_digits(X_mnist_recovered[::2100])
plt.title("Compressed", fontsize=16)
#save_fig("mnist_compression_plot")
plt.show()
```



Incremental PCA

- PCA normally requires entire dataset in memory for SVD algorithm.
- **Incremental PCA (IPCA)** splits dataset into batches.

```
# split MNIST into 100 minibatches using Numpy array_split()
# reduce MNIST down to 154 dimensions as before.
# note use of partial_fit() for each batch.

from sklearn.decomposition import IncrementalPCA

n_batches = 100
inc_pca = IncrementalPCA(n_components=154)

for X_batch in np.array_split(X_mnist, n_batches):
    print(".", end="")
    inc_pca.partial_fit(X_batch)

X_mnist_reduced_inc = inc_pca.transform(X_mnist)
```

```
................................................................
....................................
```

```
# alternative: Numpy memmap class (use binary array on disk as i
f it was in memory)

filename = "my_mnist.data"

X_mm = np.memmap(
    filename, dtype='float32', mode='write', shape=X_mnist.shape
)

X_mm[:] = X_mnist
del X_mm
```

```
X_mm = np.memmap(filename, dtype='float32', mode='readonly', sha
pe=X_mnist.shape)

batch_size = len(X_mnist) // n_batches
inc_pca = IncrementalPCA(n_components=154, batch_size=batch_size
)
inc_pca.fit(X_mm)
```

```
IncrementalPCA(batch_size=525, copy=True, n_components=154, whit
en=False)
```

```
rnd_pca = PCA(
    n_components=154,
    random_state=42,
    svd_solver="randomized")

X_reduced = rnd_pca.fit_transform(X_mnist)
```

```
import time

for n_components in (2, 10, 154):
    print("n_components =", n_components)
    regular_pca = PCA(
        n_components=n_components)
    inc_pca     = IncrementalPCA(
        n_components=154,
        batch_size=500)
    rnd_pca     = PCA(
        n_components=154,
        random_state=42,
        svd_solver="randomized")

    for pca in (regular_pca, inc_pca, rnd_pca):
        t1 = time.time()
        pca.fit(X_mnist)
        t2 = time.time()
        print(pca.__class__.__name__, t2 - t1, "seconds")
```

```
n_components = 2
PCA 1.308387279510498 seconds
IncrementalPCA 18.326093673706055 seconds
PCA 3.998342514038086 seconds
n_components = 10
PCA 1.4705824851989746 seconds
IncrementalPCA 16.598721742630005 seconds
PCA 4.156355619430542 seconds
n_components = 154
PCA 4.129154682159424 seconds
IncrementalPCA 16.597434043884277 seconds
PCA 4.0131142139434814 seconds
```

Randomized PCA

- Stochastic algorithm, quickly finds approximation of 1st d components. Dramatically faster.

```
rnd_pca = PCA(n_components=154, svd_solver="randomized")

t1 = time.time()
X_reduced = rnd_pca.fit_transform(X_mnist)
t2 = time.time()
print(t2-t1, "seconds")
```

```
4.414088487625122 seconds
```

Kernel PCA

- Use kernel trick to map instances into higher-D feature spaces. This enables non-linear classification & regression with SVMs.
- Good at preserving clusters after projecton.

```
# Below: Swiss roll reduced to 2D using 3 techniques:
# 1) linear kernel (equiv to PCA)
# 2) RBF kernel
# 3) sigmoid kernel (logistic)

from sklearn.decomposition import KernelPCA

X, t = make_swiss_roll(
    n_samples=1000,
    noise=0.2,
    random_state=42)

lin_pca = KernelPCA(
    n_components = 2,
    kernel="linear",
    fit_inverse_transform=True)

rbf_pca = KernelPCA(
    n_components = 2,
    kernel="rbf",
    gamma=0.0433,
    fit_inverse_transform=True)
```

```
sig_pca = KernelPCA(
    n_components = 2,
    kernel="sigmoid",
    gamma=0.001,
    coef0=1,
    fit_inverse_transform=True)

y = t > 6.9

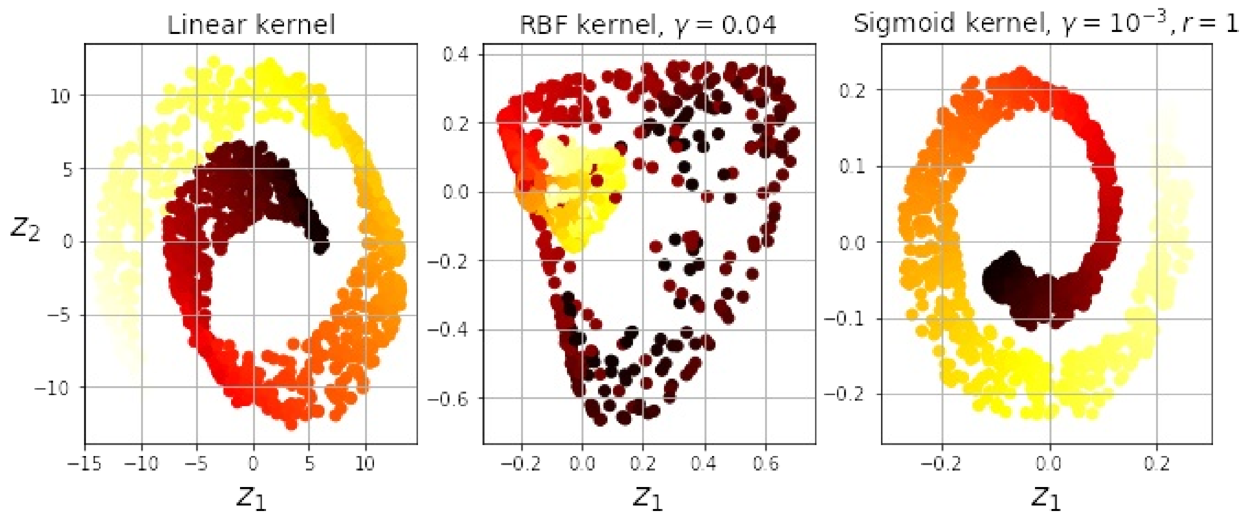
plt.figure(figsize=(11, 4))

for subplot, pca, title in (
    (131, lin_pca, "Linear kernel"),
    (132, rbf_pca, "RBF kernel, $\gamma=0.04$"),
    (133, sig_pca, "Sigmoid kernel, $\gamma=10^{-3}, r=1$")):

    X_reduced = pca.fit_transform(X)
    if subplot == 132:
        X_reduced_rbf = X_reduced

    plt.subplot(subplot)
    #plt.plot(X_reduced[y, 0], X_reduced[y, 1], "gs")
    #plt.plot(X_reduced[~y, 0], X_reduced[~y, 1], "y^")
    plt.title(title, fontsize=14)
    plt.scatter(X_reduced[:, 0], X_reduced[:, 1], c=t, cmap=plt.
cm.hot)
    plt.xlabel("$z_1$", fontsize=18)
    if subplot == 131:
        plt.ylabel("$z_2$", fontsize=18, rotation=0)
    plt.grid(True)

#save_fig("kernel_pca_plot")
plt.show()
```



Selecting a Kernel & Hyperparameters

- Dimensionality reduction = prep for supervised learning task
- Can use grid search to select kernel & params

```
from sklearn.model_selection import GridSearchCV
from sklearn.linear_model import LogisticRegression
from sklearn.pipeline import Pipeline

clf = Pipeline([
    ("kpca", KernelPCA(n_components=2)),
    ("log_reg", LogisticRegression())])

param_grid = [{
    "kpca__gamma": np.linspace(0.03, 0.05, 10),
    "kpca__kernel": ["rbf", "sigmoid"]}]

grid_search = GridSearchCV(clf, param_grid, cv=3)
grid_search.fit(X, y)

# best kernel & params?
print(grid_search.best_params_)
```

```
{'kpca__gamma': 0.043333333333333335, 'kpca__kernel': 'rbf'}
```

- Another (unsupervised approach): select kernel & params with **lowest reconstruction error**. Not as easy as with linear PCA.

```
rbf_pca = KernelPCA(
    n_components = 2,
    kernel="rbf",
    gamma=0.0433,
    fit_inverse_transform=True) # perform reconstruction

X_reduced  = rbf_pca.fit_transform(X)
X_preimage = rbf_pca.inverse_transform(X_reduced)

# return reconstruction pre-image error
from sklearn.metrics import mean_squared_error
mean_squared_error(X, X_preimage)
```

```
32.786308795766082
```

```
times_rpca = []
times_pca = []
sizes = [1000, 10000,  20000,  30000, 40000, 50000, 70000,
              100000, 200000, 500000]

for n_samples in sizes:

    X = rnd.randn(n_samples, 5)

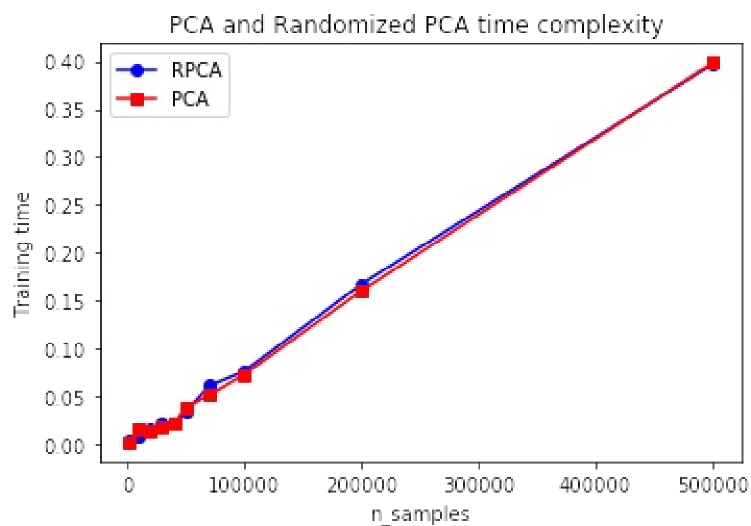
    pca = PCA(
        n_components = 2,
        random_state=42,
        svd_solver="randomized")

    t1 = time.time()
    pca.fit(X)
    t2 = time.time()
    times_rpca.append(t2 - t1)

    pca = PCA(n_components = 2)

    t1 = time.time()
    pca.fit(X)
    t2 = time.time()
    times_pca.append(t2 - t1)

plt.plot(sizes, times_rpca, "b-o", label="RPCA")
plt.plot(sizes, times_pca, "r-s", label="PCA")
plt.xlabel("n_samples")
plt.ylabel("Training time")
plt.legend(loc="upper left")
plt.title("PCA and Randomized PCA time complexity ")
plt.show()
```



LLE (Locally Linear Embedding)

- Powerful nonlinear dimensionality reduction tool
- Manifold Learning; doesn't rely on projections.
- LLE measures how each instance relates to closest neighbors, then looks for low-D representation where local relations are best preserved.

```
# Use LLE to unroll a Swiss Roll.

from sklearn.manifold import LocallyLinearEmbedding

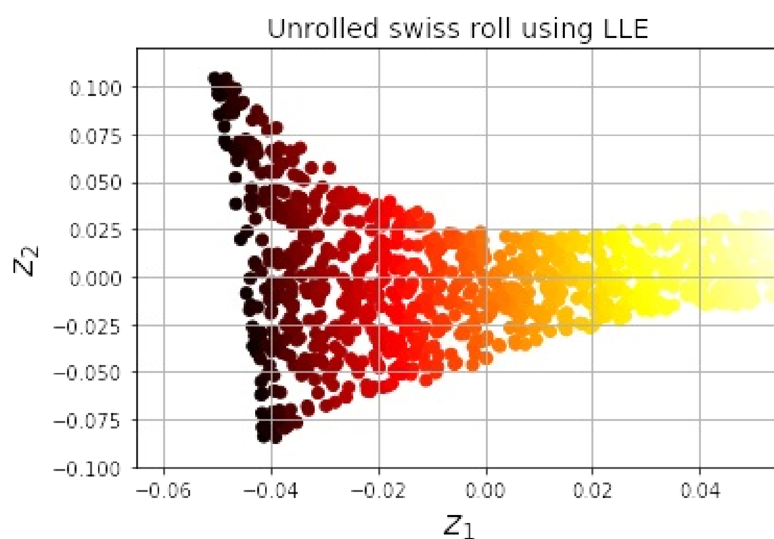
X, t = make_swiss_roll(
    n_samples=1000,
    noise=0.2,
    random_state=41)

lle = LocallyLinearEmbedding(
    n_neighbors=10,
    n_components=2,
    random_state=42)

X_reduced = lle.fit_transform(X)

plt.title("Unrolled swiss roll using LLE", fontsize=14)
plt.scatter(X_reduced[:, 0], X_reduced[:, 1], c=t, cmap=plt.cm.h
ot)
plt.xlabel("$z_1$", fontsize=18)
plt.ylabel("$z_2$", fontsize=18)
plt.axis([-0.065, 0.055, -0.1, 0.12])
plt.grid(True)

#save_fig("lle_unrolling_plot")
plt.show()
```



- 1st: For each instance, LLE finds k nearest neighbors & tries to reconstruct instance as linear function of neighbors (weights such that squared distance is minimum).
- Weight matrix W now encodes all local linear relations between instances.
- 2nd: Map instances into d-dimensional space & preserve relationship data
- Scikit computational complexity:
- finding K nearest neighbors: O(m x log(m) x n x log(k))
- weight optimization: O(m x n x k^3)
- constructing low-d representations: O(d x m^2)

MDS, Isomap, t-SNE, LDA

```
from sklearn.manifold import MDS
mds = MDS(n_components=2, random_state=42)
X_reduced_mds = mds.fit_transform(X)
```

```
from sklearn.manifold import Isomap
isomap = Isomap(n_components=2)
X_reduced_isomap = isomap.fit_transform(X)
```

```
from sklearn.manifold import TSNE
tsne = TSNE(n_components=2)
X_reduced_tsne = tsne.fit_transform(X)
```

```
from sklearn.discriminant_analysis import LinearDiscriminantAnal
ysis
lda = LinearDiscriminantAnalysis(n_components=2)
X_mnist = mnist["data"]
y_mnist = mnist["target"]
lda.fit(X_mnist, y_mnist)
X_reduced_lda = lda.transform(X_mnist)
```

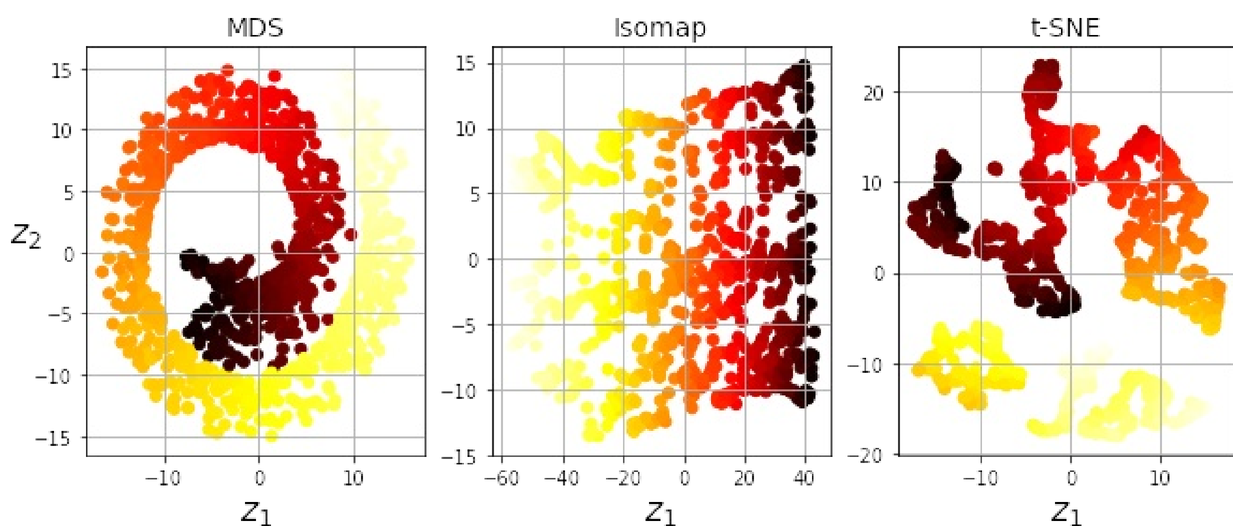
```
/home/bjpcjp/anaconda3/lib/python3.5/site-packages/sklearn/discr
iminant_analysis.py:387: UserWarning: Variables are collinear.
  warnings.warn("Variables are collinear.")
```

```
titles = ["MDS", "Isomap", "t-SNE"]

plt.figure(figsize=(11,4))

for subplot, title, X_reduced in zip((131, 132, 133), titles,
                                     (X_reduced_mds, X_reduced_i
somap, X_reduced_tsne)):
    plt.subplot(subplot)
    plt.title(title, fontsize=14)
    plt.scatter(X_reduced[:, 0], X_reduced[:, 1], c=t, cmap=plt.
cm.hot)
    plt.xlabel("$z_1$", fontsize=18)
    if subplot == 131:
        plt.ylabel("$z_2$", fontsize=18, rotation=0)
    plt.grid(True)

#save_fig("other_dim_reduction_plot")
plt.show()
```



Intro

- Graphs defined in Python, executed in C++
- Open-sourced 2015. Windows, Linux, macOS, iOS, Android
- Python API: **tensorflow.contrib.learn** (trains NNs)
- Simpler API: **tensorflow.contrib.slim** (simplifies building NNs)
- Other high-level APIs: **Keras**, **Pretty Tensor**.
- Libraries: **Caffe**, **DeepLearning4J**, **H2O**, **MXNet**, **Theano**, **Torch**
- **TensorBoard** visualization tool
- Cloud service
- Resources: home page, GitHub, StackOverflow

Installation & Test

$ cd

$ source env/bin/activate (if using virtualenv)

$ pip3 install --upgrade tensorflow (or tensorflow-gpu for GPU support)

$ python3 -c 'import tensorflow; print(tensorflow.**version**)'

```
!python3 -c 'import tensorflow; print(tensorflow.__version__)'
```

```
1.0.0
```

```
import numpy as np
```

First Graph

```
# create your first graph
import tensorflow as tf
x = tf.Variable(3, name="x")
y = tf.Variable(4, name="y")
f = x*x*y + y + 2

# run graph by opening a session

sess = tf.Session()
sess.run(x.initializer)
sess.run(y.initializer)
result = sess.run(f)
print(result)
sess.close()
```

```
42
```

```
# for repeated session "runs"

with tf.Session() as sess:
    x.initializer.run()
    y.initializer.run()
    result = f.eval()

print(result)
```

```
42
```

```
# use global_variables_initializer() to set up initialization

init = tf.global_variables_initializer()

with tf.Session() as sess:
    init.run() # actually initialize all the variables
    result = f.eval()
print(result)
```

```
42
```

```
# interactive sessions (from within Jupyter or Python shell)
# interactive sesions are auto-set as default sessions

sess = tf.InteractiveSession()
init.run()
result = f.eval()
print(result)
sess.close()
```

```
42
```

Managing Graphs

```
# any created node = added to default graph
x1 = tf.Variable(1)
x1.graph is tf.get_default_graph()
```

```
True
```

```
# handling multiple graphs

graph = tf.Graph()
with graph.as_default():
    x2 = tf.Variable(2)

x2.graph is graph, x2.graph is tf.get_default_graph()
```

```
(True, False)
```

Node Lifecycles

```
# TF finds node's dependencies & evaluates them first

w = tf.constant(3)
x = w + 2
y = x + 5
z = x * 3

# previous eval results = NOT reused. above code evals w & x twi
ce.

with tf.Session() as sess:
    print(y.eval())
    print(z.eval())

# a more efficient evaluation call:
with tf.Session() as sess:
    y_val, z_val = sess.run([y,z])
    print(y_val)
    print(z_val)
```

```
10
15
10
15
```

Linear Regression with TF

- TF ops take any number of inputs & produce any number of outputs
- Constants & variables = source ops (no inputs)
- Inputs & outputs = multidimensional "tensors" = **NumPy ndarrays** in Python API. Typically floats, can also be strings.

Below: Linear Regression on 2D arrays (California Housing dataset)

```
# (never used Numpy c_ operator before. Had to check it out.)
a = np.ones((6,1))
b = np.zeros((6,4))
c = np.ones((6,2))
np.c_[a,b,c]
```

```
array([[ 1.,  0.,  0.,  0.,  0.,  1.,  1.],
       [ 1.,  0.,  0.,  0.,  0.,  1.,  1.],
       [ 1.,  0.,  0.,  0.,  0.,  1.,  1.],
       [ 1.,  0.,  0.,  0.,  0.,  1.,  1.],
       [ 1.,  0.,  0.,  0.,  0.,  1.,  1.],
       [ 1.,  0.,  0.,  0.,  0.,  1.,  1.]])
```

```
from sklearn.datasets import fetch_california_housing
import numpy as np

housing = fetch_california_housing()
m, n = housing.data.shape

# add bias feature, x0 = 1
housing_data_plus_bias = np.c_[np.ones((m, 1)), housing.data]

print(m,n,housing_data_plus_bias.shape)
```

```
20640 8 (20640, 9)
```

```
tf.reset_default_graph()

X = tf.constant(
    housing_data_plus_bias,
    dtype=tf.float64, name="X")

print("X shape: ",X.shape)

# housing.target = 1D array. Reshape to col vector to compute th
eta.
# reshape() accepts -1 = "unspecified" for a dimension.

y = tf.constant(
    housing.target.reshape(-1, 1),
    dtype=tf.float64, name="y")

XT = tf.transpose(X)

print("XT shape: ",XT.shape)

# normal equation: theta = (XT * X)^-1 * XT * y

theta = tf.matmul(
            tf.matmul(
                tf.matrix_inverse(
                    tf.matmul(XT, X)),
                    XT),
                y)

# TF doesn't immediately run the code. It creates nodes that wil
l run with eval().
# TF will auto-run on GPU if available.

with tf.Session() as sess:
    result = theta.eval()

print("theta: \n",result)
```

```
X shape:  (20640, 9)
XT shape:  (9, 20640)
theta:
 [[ -3.69419202e+01]
 [  4.36693293e-01]
 [  9.43577803e-03]
 [ -1.07322041e-01]
 [  6.45065694e-01]
 [ -3.97638942e-06]
 [ -3.78654265e-03]
 [ -4.21314378e-01]
 [ -4.34513755e-01]]
```

```
# compare to pure NumPy
X = housing_data_plus_bias
y = housing.target.reshape(-1, 1)

theta_numpy = np.linalg.inv(X.T.dot(X)).dot(X.T).dot(y)

print("theta: \n",theta_numpy)
```

```
theta:
 [[ -3.69419202e+01]
 [  4.36693293e-01]
 [  9.43577803e-03]
 [ -1.07322041e-01]
 [  6.45065694e-01]
 [ -3.97638942e-06]
 [ -3.78654265e-03]
 [ -4.21314378e-01]
 [ -4.34513755e-01]]
```

```
# compare to Scikit
from sklearn.linear_model import LinearRegression
lin_reg = LinearRegression()

lin_reg.fit(
    housing.data,
    housing.target.reshape(-1, 1))

print("theta: \n",np.r_[
    lin_reg.intercept_.reshape(-1, 1),
    lin_reg.coef_.T])
```

```
theta:
 [[ -3.69419202e+01]
 [  4.36693293e-01]
 [  9.43577803e-03]
 [ -1.07322041e-01]
 [  6.45065694e-01]
 [ -3.97638942e-06]
 [ -3.78654265e-03]
 [ -4.21314378e-01]
 [ -4.34513755e-01]]
```

Batch Gradient Descent (instead of Normal Equation):

- Could use TF; let's use Scikit first.

```
# normalize input features first - otherwise training = much slo
wer.

from sklearn.preprocessing import StandardScaler
scaler = StandardScaler()

scaled_housing_data = scaler.fit_transform(
    housing.data)

scaled_housing_data_plus_bias = np.c_[
    np.ones((m, 1)),
    scaled_housing_data]

import pandas as pd
pd.DataFrame(scaled_housing_data_plus_bias).info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 20640 entries, 0 to 20639
Data columns (total 9 columns):
0    20640 non-null float64
1    20640 non-null float64
2    20640 non-null float64
3    20640 non-null float64
4    20640 non-null float64
5    20640 non-null float64
6    20640 non-null float64
7    20640 non-null float64
8    20640 non-null float64
dtypes: float64(9)
memory usage: 1.4 MB
```

```
print("mean (axis=0): \n",scaled_housing_data_plus_bias.mean(axi
s=0))
print("mean (axis=1): \n",scaled_housing_data_plus_bias.mean(axi
s=1))
print("mean (w/bias): \n",scaled_housing_data_plus_bias.mean())
print("data shape:    \n",scaled_housing_data_plus_bias.shape)
```

```
mean (axis=0):
 [  1.00000000e+00   6.60969987e-17   5.50808322e-18   6.6096998
7e-17
  -1.06030602e-16  -1.10161664e-17   3.44255201e-18  -1.07958431
e-15
  -8.52651283e-15]
mean (axis=1):
 [ 0.38915536  0.36424355  0.5116157  ..., -0.06612179 -0.063605
87
  0.01359031]
mean (w/bias):
 0.111111111111
data shape:
 (20640, 9)
```

Manual gradient computation

```
# batch gradient step:
# theta(next) = theta - learning_rate * MSE(theta)

tf.reset_default_graph()

n_epochs = 1000
learning_rate = 0.01

X = tf.constant(
    scaled_housing_data_plus_bias,
    dtype=tf.float32, name="X")

y = tf.constant(
    housing.target.reshape(-1, 1),
    dtype=tf.float32, name="y")

theta = tf.Variable( # tf.random_uniform = generates random tens
or
    tf.random_uniform([n+1, 1], -1.0, 1.0, seed=42),
    name="theta")
```

```
y_pred = tf.matmul(
    X, theta, name="predictions")

error       = y_pred - y
mse         = tf.reduce_mean(tf.square(error), name="mse")
gradients   = 2/m * tf.matmul(tf.transpose(X), error)
training_op = tf.assign(theta, theta - learning_rate * gradients
)

init = tf.global_variables_initializer()

with tf.Session() as sess:
    sess.run(init)

    for epoch in range(n_epochs):
        if epoch % 100 == 0: # do every 100th epoch:
            print("Epoch", epoch, "MSE =", mse.eval())
        sess.run(training_op)

    best_theta = theta.eval()

print("Best theta: \n",best_theta)
```

```
Epoch 0 MSE = 2.75443
Epoch 100 MSE = 0.632222
Epoch 200 MSE = 0.57278
Epoch 300 MSE = 0.558501
Epoch 400 MSE = 0.549069
Epoch 500 MSE = 0.542288
Epoch 600 MSE = 0.537379
Epoch 700 MSE = 0.533822
Epoch 800 MSE = 0.531243
Epoch 900 MSE = 0.529371
Best theta:
 [[  2.06855226e+00]
 [  7.74078071e-01]
 [  1.31192386e-01]
 [ -1.17845096e-01]
 [  1.64778158e-01]
 [  7.44080753e-04]
 [ -3.91945168e-02]
 [ -8.61356616e-01]
 [ -8.23479712e-01]]
```

Using autodiff

- automatically finds gradients. Note the different gradients assignment.

```
tf.reset_default_graph()

n_epochs = 1000
learning_rate = 0.01

X = tf.constant(
    scaled_housing_data_plus_bias,
    dtype=tf.float32, name="X")

y = tf.constant(
    housing.target.reshape(-1, 1),
    dtype=tf.float32, name="y")
```

```
theta = tf.Variable(
    tf.random_uniform([n + 1, 1], -1.0, 1.0,
                      seed=42), name="theta")

y_pred = tf.matmul(
    X, theta, name="predictions")

error        = y_pred - y
mse          = tf.reduce_mean(tf.square(error), name="mse")

# AutoDiff to the rescue
# creates list of ops, one/variable, to find gradients per varia
ble
gradients    = tf.gradients(mse, [theta])[0]
#

training_op = tf.assign(theta, theta - learning_rate * gradients
)

init = tf.global_variables_initializer()

with tf.Session() as sess:
    sess.run(init)

    for epoch in range(n_epochs):
        if epoch % 100 == 0:
            print("Epoch", epoch, "MSE =", mse.eval())
        sess.run(training_op)

    best_theta = theta.eval()

print("Best theta: \n", best_theta)
```

```
Epoch 0 MSE = 2.75443
Epoch 100 MSE = 0.632222
Epoch 200 MSE = 0.57278
Epoch 300 MSE = 0.558501
Epoch 400 MSE = 0.549069
Epoch 500 MSE = 0.542288
Epoch 600 MSE = 0.537379
Epoch 700 MSE = 0.533822
Epoch 800 MSE = 0.531243
Epoch 900 MSE = 0.529371
Best theta:
 [[  2.06855249e+00]
 [  7.74078071e-01]
 [  1.31192386e-01]
 [ -1.17845066e-01]
 [  1.64778143e-01]
 [  7.44078017e-04]
 [ -3.91945094e-02]
 [ -8.61356676e-01]
 [ -8.23479772e-01]]
```

Four ways to run autodiff - see Appendix D

- reverse-mode (default): best for many inputs, few outputs
- symbolic diff: high accuracy
- forward mode: high accuracy
- numerical diff: low accuracy, but trivial to implement

Using a predefined optimizer (Gradient Descent)

```
tf.reset_default_graph()

n_epochs = 1000
learning_rate = 0.01

X = tf.constant(
    scaled_housing_data_plus_bias,
```

```
    dtype=tf.float32, name="X")

y = tf.constant(
    housing.target.reshape(-1, 1),
    dtype=tf.float32, name="y")

theta = tf.Variable(
    tf.random_uniform([n + 1, 1], -1.0, 1.0, seed=42),
    name="theta")

y_pred = tf.matmul(
    X, theta, name="predictions")

error       = y_pred - y
mse         = tf.reduce_mean(tf.square(error), name="mse")

#####
optimizer   = tf.train.GradientDescentOptimizer(learning_rate=le
arning_rate)
training_op = optimizer.minimize(mse)
#####

init = tf.global_variables_initializer()

with tf.Session() as sess:
    sess.run(init)

    for epoch in range(n_epochs):
        if epoch % 100 == 0:
            print("Epoch", epoch, "MSE =", mse.eval())
        sess.run(training_op)

    best_theta = theta.eval()

print("Best theta:\n", best_theta)
```

```
Epoch 0 MSE = 2.75443
Epoch 100 MSE = 0.632222
Epoch 200 MSE = 0.57278
Epoch 300 MSE = 0.558501
Epoch 400 MSE = 0.549069
Epoch 500 MSE = 0.542288
Epoch 600 MSE = 0.537379
Epoch 700 MSE = 0.533822
Epoch 800 MSE = 0.531243
Epoch 900 MSE = 0.529371
Best theta:
 [[  2.06855249e+00]
 [  7.74078071e-01]
 [  1.31192386e-01]
 [ -1.17845066e-01]
 [  1.64778143e-01]
 [  7.44078017e-04]
 [ -3.91945094e-02]
 [ -8.61356676e-01]
 [ -8.23479772e-01]]
```

Using a predefined optimizer (Momentum)

```
tf.reset_default_graph()

n_epochs = 1000
learning_rate = 0.01

X = tf.constant(
    scaled_housing_data_plus_bias,
    dtype=tf.float32, name="X")

y = tf.constant(
    housing.target.reshape(-1, 1),
    dtype=tf.float32, name="y")

theta = tf.Variable(
    tf.random_uniform([n + 1, 1], -1.0, 1.0, seed=42),
    name="theta")

y_pred      = tf.matmul(X, theta, name="predictions")
error       = y_pred - y
mse         = tf.reduce_mean(tf.square(error), name="mse")
#####
optimizer   = tf.train.MomentumOptimizer(
    learning_rate=learning_rate,
    momentum=0.25)
#####
training_op = optimizer.minimize(mse)

init = tf.global_variables_initializer()

with tf.Session() as sess:
    sess.run(init)

    for epoch in range(n_epochs):
        sess.run(training_op)

    best_theta = theta.eval()

print("Best theta:\n", best_theta)
```

```
Best theta:
 [[  2.06855392e+00]
 [  7.94067979e-01]
 [  1.25333667e-01]
 [ -1.73580602e-01]
 [  2.18767926e-01]
 [ -1.64708309e-03]
 [ -3.91250364e-02]
 [ -8.85289013e-01]
 [ -8.50607991e-01]]
```

Training & Data Feeds

- Goal: modify previous code for Minibatch gradient descent
- Best practice: *placeholder* nodes (no computation, just data output)

```
A = tf.placeholder(tf.float32, shape=(None, 3))
B = A + 5

with tf.Session() as sess:
    B_val_1 = B.eval(
        feed_dict={A: [[1, 2, 3]]})

    B_val_2 = B.eval(
        feed_dict={A: [[4, 5, 6], [7, 8, 9]]})

print(B_val_1, "\n", B_val_2)
```

```
[[ 6.  7.  8.]]
 [[  9.  10.  11.]
 [ 12.  13.  14.]]
```

```
# definition phase: change X,y to placeholder nodes
tf.reset_default_graph()

n_epochs = 1000
learning_rate = 0.01

##########

X = tf.placeholder(tf.float32, shape=(None, n+1), name="X")
y = tf.placeholder(tf.float32, shape=(None, 1),   name="y")

##########

theta = tf.Variable(
    tf.random_uniform([n+1, 1], -1.0, 1.0, seed=42),
    name="theta")

y_pred = tf.matmul(
    X, theta, name="predictions")

error  = y_pred - y

mse = tf.reduce_mean(tf.square(error), name="mse")

optimizer = tf.train.GradientDescentOptimizer(
    learning_rate=learning_rate)

training_op = optimizer.minimize(mse)

init = tf.global_variables_initializer()
```

```
# execution phase: fetch minibatches one-by-one.
# use feed_dict to provide values to dependent nodes

import numpy.random as rnd

n_epochs = 10
batch_size = 100
n_batches = int(np.ceil(m / batch_size))

print("m: ",m,"\n","n_batches: ",n_batches,"\n")

def fetch_batch(epoch, batch_index, batch_size):

    rnd.seed(epoch * n_batches + batch_index)
    indices = rnd.randint(m, size=batch_size)

    X_batch = scaled_housing_data_plus_bias[indices]
    y_batch = housing.target.reshape(-1, 1)[indices]

    return X_batch, y_batch

with tf.Session() as sess:
    sess.run(init)

    for epoch in range(n_epochs):
        for batch_index in range(n_batches):

            X_batch, y_batch = fetch_batch(
                epoch, batch_index, batch_size)

            sess.run(
                training_op,
                feed_dict={X: X_batch, y: y_batch})

    best_theta = theta.eval()

print("Best theta: \n",best_theta)
```

```
m:  20640
 n_batches:  207

Best theta:
 [[ 2.07001591]
 [ 0.82045609]
 [ 0.1173173 ]
 [-0.22739051]
 [ 0.31134021]
 [ 0.00353193]
 [-0.01126994]
 [-0.91643935]
 [-0.87950081]]
```

Model Save/Restore

- Use a *saver* node once construction is complete.
- call **save()** method - pass it session and filepath info.

```
tf.reset_default_graph()

n_epochs = 1000
learning_rate = 0.01

X = tf.constant(
    scaled_housing_data_plus_bias,
    dtype=tf.float32, name="X")

y = tf.constant(
    housing.target.reshape(-1, 1),
    dtype=tf.float32, name="y")

theta = tf.Variable(
    tf.random_uniform([n+1, 1], -1.0, 1.0, seed=42),
    name="theta")

y_pred = tf.matmul(
    X, theta, name="predictions")

error = y_pred - y

mse = tf.reduce_mean(
    tf.square(error),
    name="mse")

optimizer = tf.train.GradientDescentOptimizer(
    learning_rate=learning_rate)

training_op = optimizer.minimize(mse)

init = tf.global_variables_initializer()

saver = tf.train.Saver()
# can specify which vars to save:
# saver = tf.train.Saver({"weights": theta})
```

```
with tf.Session() as sess:
    sess.run(init)

    for epoch in range(n_epochs):
        if epoch % 100 == 0:
            print("Epoch", epoch, "MSE =", mse.eval())
            save_path = saver.save(sess, "/tmp/my_model.ckpt")
        sess.run(training_op)

    best_theta = theta.eval()
    save_path = saver.save(sess, "my_model_final.ckpt")

print("Best theta:\n",best_theta)

# model restoration:
# 1) create Saver at end of construction phase
# 2) call saver.restore() at start of execution
```

```
Epoch 0 MSE = 2.75443
Epoch 100 MSE = 0.632222
Epoch 200 MSE = 0.57278
Epoch 300 MSE = 0.558501
Epoch 400 MSE = 0.549069
Epoch 500 MSE = 0.542288
Epoch 600 MSE = 0.537379
Epoch 700 MSE = 0.533822
Epoch 800 MSE = 0.531243
Epoch 900 MSE = 0.529371
Best theta:
 [[  2.06855249e+00]
 [  7.74078071e-01]
 [  1.31192386e-01]
 [ -1.17845066e-01]
 [  1.64778143e-01]
 [  7.44078017e-04]
 [ -3.91945094e-02]
 [ -8.61356676e-01]
 [ -8.23479772e-01]]
```

```
!cat checkpoint
```

```
model_checkpoint_path: "my_model_final.ckpt"

all_model_checkpoint_paths: "/tmp/my_model.ckpt"

all_model_checkpoint_paths: "my_model_final.ckpt"
```

Visualization - inside Jupyter

```
from IPython.display import clear_output, Image, display, HTML

def strip_consts(graph_def, max_const_size=32):
    """Strip large constant values from graph_def."""
    strip_def = tf.GraphDef()
    for n0 in graph_def.node:
        n = strip_def.node.add()
        n.MergeFrom(n0)
        if n.op == 'Const':
            tensor = n.attr['value'].tensor
            size = len(tensor.tensor_content)
            if size > max_const_size:
                tensor.tensor_content = b"<stripped %d bytes>"%s
ize
    return strip_def

def show_graph(graph_def, max_const_size=32):
    """Visualize TensorFlow graph."""
    if hasattr(graph_def, 'as_graph_def'):
        graph_def = graph_def.as_graph_def()

    strip_def = strip_consts(graph_def, max_const_size=max_const
_size)

    code = """
        <script>
          function load() {{
```

```
              document.getElementById("{id}").pbtxt = {data};
            }}
          </script>
          <link rel="import" href="https://tensorboard.appspot.com
/tf-graph-basic.build.html" onload=load()>
          <div style="height:600px">
            <tf-graph-basic id="{id}"></tf-graph-basic>
          </div>
    """.format(data=repr(str(strip_def)), id='graph'+str(np.rand
om.rand()))

    # original was width=1200px, height=620px
    iframe = """
        <iframe seamless style="width:1200px;height:620px;border
:0" srcdoc="{}"></iframe>
    """.format(code.replace('"', '&quot;'))
    display(HTML(iframe))
```

```
show_graph(tf.get_default_graph())
```

```
    <iframe seamless style="width:1200px;height:620px;border:0"
srcdoc="
    <script>
      function load() {
        document.getElementById(&quot;graph0.1784179106002547&qu
ot;).pbtxt = 'node {\n  name: &quot;X&quot;\n  op: &quot;Const&q
uot;\n  attr {\n    key: &quot;dtype&quot;\n    value {\n      t
ype: DT_FLOAT\n    }\n  }\n  attr {\n    key: &quot;value&quot;\
n    value {\n      tensor {\n        dtype: DT_FLOAT\n        t
ensor_shape {\n          dim {\n            size: 20640\n
   }\n          dim {\n            size: 9\n          }\n
 }\n        tensor_content: &quot;<stripped 743040 bytes>&quot;\
n      }\n    }\n  }\n}\nnode {\n  name: &quot;y&quot;\n  op: &q
uot;Const&quot;\n  attr {\n    key: &quot;dtype&quot;\n    value
 {\n      type: DT_FLOAT\n    }\n  }\n  attr {\n    key: &quot;v
alue&quot;\n    value {\n      tensor {\n        dtype: DT_FLOAT
\n        tensor_shape {\n          dim {\n            size: 206
40\n          }\n          dim {\n            size: 1\n
```

```
 }\n        }\n        tensor_content: &quot;<stripped 82560 byt
es>&quot;\n      }\n    }\n  }\n}\nnode {\n  name: &quot;random_
uniform/shape&quot;\n  op: &quot;Const&quot;\n  attr {\n    key:
 &quot;dtype&quot;\n    value {\n      type: DT_INT32\n    }\n
}\n  attr {\n    key: &quot;value&quot;\n    value {\n      tens
or {\n        dtype: DT_INT32\n        tensor_shape {\n
 dim {\n            size: 2\n          }\n        }\n        ten
sor_content: &quot;\\t\\000\\000\\000\\001\\000\\000\\000&quot;\
n      }\n    }\n  }\n}\nnode {\n  name: &quot;random_uniform/mi
n&quot;\n  op: &quot;Const&quot;\n  attr {\n    key: &quot;dtype
&quot;\n    value {\n      type: DT_FLOAT\n    }\n  }\n  attr {\
n    key: &quot;value&quot;\n    value {\n      tensor {\n
  dtype: DT_FLOAT\n        tensor_shape {\n        }\n        fl
oat_val: -1.0\n      }\n    }\n  }\n}\nnode {\n  name: &quot;ran
dom_uniform/max&quot;\n  op: &quot;Const&quot;\n  attr {\n    ke
y: &quot;dtype&quot;\n    value {\n      type: DT_FLOAT\n    }\n
  }\n  attr {\n    key: &quot;value&quot;\n    value {\n      te
nsor {\n        dtype: DT_FLOAT\n        tensor_shape {\n
 }\n        float_val: 1.0\n      }\n    }\n  }\n}\nnode {\n  na
me: &quot;random_uniform/RandomUniform&quot;\n  op: &quot;Random
Uniform&quot;\n  input: &quot;random_uniform/shape&quot;\n  attr
 {\n    key: &quot;T&quot;\n    value {\n      type: DT_INT32\n
   }\n  }\n  attr {\n    key: &quot;dtype&quot;\n    value {\n
    type: DT_FLOAT\n    }\n  }\n  attr {\n    key: &quot;seed&qu
ot;\n    value {\n      i: 87654321\n    }\n  }\n  attr {\n    k
ey: &quot;seed2&quot;\n    value {\n      i: 42\n    }\n  }\n}\n
node {\n  name: &quot;random_uniform/sub&quot;\n  op: &quot;Sub&
quot;\n  input: &quot;random_uniform/max&quot;\n  input: &quot;r
andom_uniform/min&quot;\n  attr {\n    key: &quot;T&quot;\n    v
alue {\n      type: DT_FLOAT\n    }\n  }\n}\nnode {\n  name: &qu
ot;random_uniform/mul&quot;\n  op: &quot;Mul&quot;\n  input: &qu
ot;random_uniform/RandomUniform&quot;\n  input: &quot;random_uni
form/sub&quot;\n  attr {\n    key: &quot;T&quot;\n    value {\n
     type: DT_FLOAT\n    }\n  }\n}\nnode {\n  name: &quot;random
_uniform&quot;\n  op: &quot;Add&quot;\n  input: &quot;random_uni
form/mul&quot;\n  input: &quot;random_uniform/min&quot;\n  attr
{\n    key: &quot;T&quot;\n    value {\n      type: DT_FLOAT\n
  }\n  }\n}\nnode {\n  name: &quot;theta&quot;\n  op: &quot;Vari
ableV2&quot;\n  attr {\n    key: &quot;container&quot;\n    valu
e {\n      s: &quot;&quot;\n    }\n  }\n  attr {\n    key: &quot
```

```
;dtype&quot;\n    value {\n      type: DT_FLOAT\n    }\n  }\n  a
ttr {\n    key: &quot;shape&quot;\n    value {\n      shape {\n
       dim {\n          size: 9\n        }\n        dim {\n
     size: 1\n        }\n      }\n    }\n  }\n  attr {\n    key:
 &quot;shared_name&quot;\n    value {\n      s: &quot;&quot;\n
  }\n  }\n}\nnode {\n  name: &quot;theta/Assign&quot;\n  op: &qu
ot;Assign&quot;\n  input: &quot;theta&quot;\n  input: &quot;rand
om_uniform&quot;\n  attr {\n    key: &quot;T&quot;\n    value {\
n      type: DT_FLOAT\n    }\n  }\n  attr {\n    key: &quot;_cla
ss&quot;\n    value {\n      list {\n        s: &quot;loc:@theta
&quot;\n      }\n    }\n  }\n  attr {\n    key: &quot;use_lockin
g&quot;\n    value {\n      b: true\n    }\n  }\n  attr {\n    k
ey: &quot;validate_shape&quot;\n    value {\n      b: true\n
}\n  }\n}\nnode {\n  name: &quot;theta/read&quot;\n  op: &quot;I
dentity&quot;\n  input: &quot;theta&quot;\n  attr {\n    key: &q
uot;T&quot;\n    value {\n      type: DT_FLOAT\n    }\n  }\n  at
tr {\n    key: &quot;_class&quot;\n    value {\n      list {\n
      s: &quot;loc:@theta&quot;\n      }\n    }\n  }\n}\nnode {\
n  name: &quot;predictions&quot;\n  op: &quot;MatMul&quot;\n  in
put: &quot;X&quot;\n  input: &quot;theta/read&quot;\n  attr {\n
   key: &quot;T&quot;\n    value {\n      type: DT_FLOAT\n    }\
n  }\n  attr {\n    key: &quot;transpose_a&quot;\n    value {\n
     b: false\n    }\n  }\n  attr {\n    key: &quot;transpose_b&
quot;\n    value {\n      b: false\n    }\n  }\n}\nnode {\n  nam
e: &quot;sub&quot;\n  op: &quot;Sub&quot;\n  input: &quot;predic
tions&quot;\n  input: &quot;y&quot;\n  attr {\n    key: &quot;T&
quot;\n    value {\n      type: DT_FLOAT\n    }\n  }\n}\nnode {\
n  name: &quot;Square&quot;\n  op: &quot;Square&quot;\n  input:
&quot;sub&quot;\n  attr {\n    key: &quot;T&quot;\n    value {\n
      type: DT_FLOAT\n    }\n  }\n}\nnode {\n  name: &quot;Const
&quot;\n  op: &quot;Const&quot;\n  attr {\n    key: &quot;dtype&
quot;\n    value {\n      type: DT_INT32\n    }\n  }\n  attr {\n
    key: &quot;value&quot;\n    value {\n      tensor {\n
 dtype: DT_INT32\n        tensor_shape {\n          dim {\n
       size: 2\n          }\n        }\n        tensor_content:
&quot;\\000\\000\\000\\000\\001\\000\\000\\000&quot;\n      }\n
   }\n  }\n}\nnode {\n  name: &quot;mse&quot;\n  op: &quot;Mean&
quot;\n  input: &quot;Square&quot;\n  input: &quot;Const&quot;\n
  attr {\n    key: &quot;T&quot;\n    value {\n      type: DT_FL
OAT\n    }\n  }\n  attr {\n    key: &quot;Tidx&quot;\n    value
```

```
{\n      type: DT_INT32\n    }\n  }\n  attr {\n    key: &quot;ke
ep_dims&quot;\n    value {\n      b: false\n    }\n  }\n}\nnode
{\n  name: &quot;gradients/Shape&quot;\n  op: &quot;Const&quot;\
n  attr {\n    key: &quot;dtype&quot;\n    value {\n      type:
DT_INT32\n    }\n  }\n  attr {\n    key: &quot;value&quot;\n
value {\n      tensor {\n        dtype: DT_INT32\n        tensor
_shape {\n          dim {\n          }\n        }\n      }\n
}\n  }\n}\nnode {\n  name: &quot;gradients/Const&quot;\n  op: &q
uot;Const&quot;\n  attr {\n    key: &quot;dtype&quot;\n    value
 {\n      type: DT_FLOAT\n    }\n  }\n  attr {\n    key: &quot;v
alue&quot;\n    value {\n      tensor {\n        dtype: DT_FLOAT
\n        tensor_shape {\n        }\n        float_val: 1.0\n
   }\n    }\n  }\n}\nnode {\n  name: &quot;gradients/Fill&quot;\
n  op: &quot;Fill&quot;\n  input: &quot;gradients/Shape&quot;\n
 input: &quot;gradients/Const&quot;\n  attr {\n    key: &quot;T&
quot;\n    value {\n      type: DT_FLOAT\n    }\n  }\n}\nnode {\
n  name: &quot;gradients/mse_grad/Reshape/shape&quot;\n  op: &qu
ot;Const&quot;\n  attr {\n    key: &quot;dtype&quot;\n    value
{\n      type: DT_INT32\n    }\n  }\n  attr {\n    key: &quot;va
lue&quot;\n    value {\n      tensor {\n        dtype: DT_INT32\
n        tensor_shape {\n          dim {\n            size: 2\n
         }\n        }\n        tensor_content: &quot;\\001\\000\
\000\\000\\001\\000\\000\\000&quot;\n      }\n    }\n  }\n}\nnod
e {\n  name: &quot;gradients/mse_grad/Reshape&quot;\n  op: &quot
;Reshape&quot;\n  input: &quot;gradients/Fill&quot;\n  input: &q
uot;gradients/mse_grad/Reshape/shape&quot;\n  attr {\n    key: &
quot;T&quot;\n    value {\n      type: DT_FLOAT\n    }\n  }\n  a
ttr {\n    key: &quot;Tshape&quot;\n    value {\n      type: DT_
INT32\n    }\n  }\n}\nnode {\n  name: &quot;gradients/mse_grad/T
ile/multiples&quot;\n  op: &quot;Const&quot;\n  attr {\n    key:
 &quot;dtype&quot;\n    value {\n      type: DT_INT32\n    }\n
}\n  attr {\n    key: &quot;value&quot;\n    value {\n      tens
or {\n        dtype: DT_INT32\n        tensor_shape {\n
 dim {\n            size: 2\n          }\n        }\n        ten
sor_content: &quot;\\240P\\000\\000\\001\\000\\000\\000&quot;\n
     }\n    }\n  }\n}\nnode {\n  name: &quot;gradients/mse_grad/
Tile&quot;\n  op: &quot;Tile&quot;\n  input: &quot;gradients/mse
_grad/Reshape&quot;\n  input: &quot;gradients/mse_grad/Tile/mult
iples&quot;\n  attr {\n    key: &quot;T&quot;\n    value {\n
  type: DT_FLOAT\n    }\n  }\n  attr {\n    key: &quot;Tmultiple
```

```
s&quot;\n    value {\n      type: DT_INT32\n    }\n  }\n}\nnode
{\n  name: &quot;gradients/mse_grad/Shape&quot;\n  op: &quot;Con
st&quot;\n  attr {\n    key: &quot;dtype&quot;\n    value {\n
   type: DT_INT32\n    }\n  }\n  attr {\n    key: &quot;value&qu
ot;\n    value {\n      tensor {\n        dtype: DT_INT32\n
   tensor_shape {\n          dim {\n            size: 2\n
   }\n        }\n        tensor_content: &quot;\\240P\\000\\000\
\001\\000\\000\\000&quot;\n      }\n    }\n  }\n}\nnode {\n  nam
e: &quot;gradients/mse_grad/Shape_1&quot;\n  op: &quot;Const&quo
t;\n  attr {\n    key: &quot;dtype&quot;\n    value {\n      typ
e: DT_INT32\n    }\n  }\n  attr {\n    key: &quot;value&quot;\n
   value {\n      tensor {\n        dtype: DT_INT32\n        ten
sor_shape {\n          dim {\n          }\n        }\n      }\n
   }\n  }\n}\nnode {\n  name: &quot;gradients/mse_grad/Const&quo
t;\n  op: &quot;Const&quot;\n  attr {\n    key: &quot;dtype&quot
;\n    value {\n      type: DT_INT32\n    }\n  }\n  attr {\n
key: &quot;value&quot;\n    value {\n      tensor {\n        dty
pe: DT_INT32\n        tensor_shape {\n          dim {\n
   size: 1\n          }\n        }\n        int_val: 0\n      }\
n    }\n  }\n}\nnode {\n  name: &quot;gradients/mse_grad/Prod&qu
ot;\n  op: &quot;Prod&quot;\n  input: &quot;gradients/mse_grad/S
hape&quot;\n  input: &quot;gradients/mse_grad/Const&quot;\n  att
r {\n    key: &quot;T&quot;\n    value {\n      type: DT_INT32\n
    }\n  }\n  attr {\n    key: &quot;Tidx&quot;\n    value {\n
    type: DT_INT32\n    }\n  }\n  attr {\n    key: &quot;keep_di
ms&quot;\n    value {\n      b: false\n    }\n  }\n}\nnode {\n
name: &quot;gradients/mse_grad/Const_1&quot;\n  op: &quot;Const&
quot;\n  attr {\n    key: &quot;dtype&quot;\n    value {\n
type: DT_INT32\n    }\n  }\n  attr {\n    key: &quot;value&quot;
\n    value {\n      tensor {\n        dtype: DT_INT32\n
tensor_shape {\n          dim {\n            size: 1\n
}\n        }\n        int_val: 0\n      }\n    }\n  }\n}\nnode {
\n  name: &quot;gradients/mse_grad/Prod_1&quot;\n  op: &quot;Pro
d&quot;\n  input: &quot;gradients/mse_grad/Shape_1&quot;\n  inpu
t: &quot;gradients/mse_grad/Const_1&quot;\n  attr {\n    key: &q
uot;T&quot;\n    value {\n      type: DT_INT32\n    }\n  }\n  at
tr {\n    key: &quot;Tidx&quot;\n    value {\n      type: DT_INT
32\n    }\n  }\n  attr {\n    key: &quot;keep_dims&quot;\n    va
lue {\n      b: false\n    }\n  }\n}\nnode {\n  name: &quot;grad
ients/mse_grad/Maximum/y&quot;\n  op: &quot;Const&quot;\n  attr
```

```
{\n    key: &quot;dtype&quot;\n    value {\n      type: DT_INT32
\n    }\n  }\n  attr {\n    key: &quot;value&quot;\n    value {\
n      tensor {\n        dtype: DT_INT32\n        tensor_shape {
\n        }\n        int_val: 1\n      }\n    }\n  }\n}\nnode {\
n  name: &quot;gradients/mse_grad/Maximum&quot;\n  op: &quot;Max
imum&quot;\n  input: &quot;gradients/mse_grad/Prod_1&quot;\n  in
put: &quot;gradients/mse_grad/Maximum/y&quot;\n  attr {\n    key
: &quot;T&quot;\n    value {\n      type: DT_INT32\n    }\n  }\n
}\nnode {\n  name: &quot;gradients/mse_grad/floordiv&quot;\n  op
: &quot;FloorDiv&quot;\n  input: &quot;gradients/mse_grad/Prod&q
uot;\n  input: &quot;gradients/mse_grad/Maximum&quot;\n  attr {\
n    key: &quot;T&quot;\n    value {\n      type: DT_INT32\n    
}\n  }\n}\nnode {\n  name: &quot;gradients/mse_grad/Cast&quot;\n
  op: &quot;Cast&quot;\n  input: &quot;gradients/mse_grad/floord
iv&quot;\n  attr {\n    key: &quot;DstT&quot;\n    value {\n    
  type: DT_FLOAT\n    }\n  }\n  attr {\n    key: &quot;SrcT&quot
;\n    value {\n      type: DT_INT32\n    }\n  }\n}\nnode {\n  n
ame: &quot;gradients/mse_grad/truediv&quot;\n  op: &quot;RealDiv
&quot;\n  input: &quot;gradients/mse_grad/Tile&quot;\n  input: &
quot;gradients/mse_grad/Cast&quot;\n  attr {\n    key: &quot;T&q
uot;\n    value {\n      type: DT_FLOAT\n    }\n  }\n}\nnode {\n
  name: &quot;gradients/Square_grad/mul/x&quot;\n  op: &quot;Con
st&quot;\n  input: &quot;^gradients/mse_grad/truediv&quot;\n  at
tr {\n    key: &quot;dtype&quot;\n    value {\n      type: DT_FL
OAT\n    }\n  }\n  attr {\n    key: &quot;value&quot;\n    value
 {\n      tensor {\n        dtype: DT_FLOAT\n        tensor_shap
e {\n        }\n        float_val: 2.0\n      }\n    }\n  }\n}\n
node {\n  name: &quot;gradients/Square_grad/mul&quot;\n  op: &qu
ot;Mul&quot;\n  input: &quot;gradients/Square_grad/mul/x&quot;\n
  input: &quot;sub&quot;\n  attr {\n    key: &quot;T&quot;\n    
value {\n      type: DT_FLOAT\n    }\n  }\n}\nnode {\n  name: &q
uot;gradients/Square_grad/mul_1&quot;\n  op: &quot;Mul&quot;\n  
input: &quot;gradients/mse_grad/truediv&quot;\n  input: &quot;gr
adients/Square_grad/mul&quot;\n  attr {\n    key: &quot;T&quot;\
n    value {\n      type: DT_FLOAT\n    }\n  }\n}\nnode {\n  nam
e: &quot;gradients/sub_grad/Shape&quot;\n  op: &quot;Const&quot;
\n  attr {\n    key: &quot;dtype&quot;\n    value {\n      type:
 DT_INT32\n    }\n  }\n  attr {\n    key: &quot;value&quot;\n   
 value {\n      tensor {\n        dtype: DT_INT32\n        tenso
r_shape {\n          dim {\n            size: 2\n          }\n
```

```
      }\n        tensor_content: &quot;\\240P\\000\\000\\001\\00
0\\000\\000&quot;\n      }\n    }\n  }\n}\nnode {\n  name: &quot
;gradients/sub_grad/Shape_1&quot;\n  op: &quot;Const&quot;\n  at
tr {\n    key: &quot;dtype&quot;\n    value {\n      type: DT_IN
T32\n    }\n  }\n  attr {\n    key: &quot;value&quot;\n    value
 {\n      tensor {\n        dtype: DT_INT32\n        tensor_shap
e {\n          dim {\n            size: 2\n          }\n
}\n        tensor_content: &quot;\\240P\\000\\000\\001\\000\\000
\\000&quot;\n      }\n    }\n  }\n}\nnode {\n  name: &quot;gradi
ents/sub_grad/BroadcastGradientArgs&quot;\n  op: &quot;Broadcast
GradientArgs&quot;\n  input: &quot;gradients/sub_grad/Shape&quot
;\n  input: &quot;gradients/sub_grad/Shape_1&quot;\n  attr {\n
  key: &quot;T&quot;\n    value {\n      type: DT_INT32\n    }\n
  }\n}\nnode {\n  name: &quot;gradients/sub_grad/Sum&quot;\n  op
: &quot;Sum&quot;\n  input: &quot;gradients/Square_grad/mul_1&qu
ot;\n  input: &quot;gradients/sub_grad/BroadcastGradientArgs&quo
t;\n  attr {\n    key: &quot;T&quot;\n    value {\n      type: D
T_FLOAT\n    }\n  }\n  attr {\n    key: &quot;Tidx&quot;\n    va
lue {\n      type: DT_INT32\n    }\n  }\n  attr {\n    key: &quo
t;keep_dims&quot;\n    value {\n      b: false\n    }\n  }\n}\nn
ode {\n  name: &quot;gradients/sub_grad/Reshape&quot;\n  op: &qu
ot;Reshape&quot;\n  input: &quot;gradients/sub_grad/Sum&quot;\n
 input: &quot;gradients/sub_grad/Shape&quot;\n  attr {\n    key:
 &quot;T&quot;\n    value {\n      type: DT_FLOAT\n    }\n  }\n
 attr {\n    key: &quot;Tshape&quot;\n    value {\n      type: D
T_INT32\n    }\n  }\n}\nnode {\n  name: &quot;gradients/sub_grad
/Sum_1&quot;\n  op: &quot;Sum&quot;\n  input: &quot;gradients/Sq
uare_grad/mul_1&quot;\n  input: &quot;gradients/sub_grad/Broadca
stGradientArgs:1&quot;\n  attr {\n    key: &quot;T&quot;\n    va
lue {\n      type: DT_FLOAT\n    }\n  }\n  attr {\n    key: &quo
t;Tidx&quot;\n    value {\n      type: DT_INT32\n    }\n  }\n  a
ttr {\n    key: &quot;keep_dims&quot;\n    value {\n      b: fal
se\n    }\n  }\n}\nnode {\n  name: &quot;gradients/sub_grad/Neg&
quot;\n  op: &quot;Neg&quot;\n  input: &quot;gradients/sub_grad/
Sum_1&quot;\n  attr {\n    key: &quot;T&quot;\n    value {\n
  type: DT_FLOAT\n    }\n  }\n}\nnode {\n  name: &quot;gradients
/sub_grad/Reshape_1&quot;\n  op: &quot;Reshape&quot;\n  input: &
quot;gradients/sub_grad/Neg&quot;\n  input: &quot;gradients/sub_
grad/Shape_1&quot;\n  attr {\n    key: &quot;T&quot;\n    value
{\n      type: DT_FLOAT\n    }\n  }\n  attr {\n    key: &quot;Ts
```

```
hape&quot;\n    value {\n      type: DT_INT32\n    }\n  }\n}\nno
de {\n  name: &quot;gradients/sub_grad/tuple/group_deps&quot;\n
 op: &quot;NoOp&quot;\n  input: &quot;^gradients/sub_grad/Reshap
e&quot;\n  input: &quot;^gradients/sub_grad/Reshape_1&quot;\n}\n
node {\n  name: &quot;gradients/sub_grad/tuple/control_dependenc
y&quot;\n  op: &quot;Identity&quot;\n  input: &quot;gradients/su
b_grad/Reshape&quot;\n  input: &quot;^gradients/sub_grad/tuple/g
roup_deps&quot;\n  attr {\n    key: &quot;T&quot;\n    value {\n
      type: DT_FLOAT\n    }\n  }\n  attr {\n    key: &quot;_clas
s&quot;\n    value {\n      list {\n        s: &quot;loc:@gradie
nts/sub_grad/Reshape&quot;\n      }\n    }\n  }\n}\nnode {\n  na
me: &quot;gradients/sub_grad/tuple/control_dependency_1&quot;\n
 op: &quot;Identity&quot;\n  input: &quot;gradients/sub_grad/Res
hape_1&quot;\n  input: &quot;^gradients/sub_grad/tuple/group_dep
s&quot;\n  attr {\n    key: &quot;T&quot;\n    value {\n      ty
pe: DT_FLOAT\n    }\n  }\n  attr {\n    key: &quot;_class&quot;\
n    value {\n      list {\n        s: &quot;loc:@gradients/sub_
grad/Reshape_1&quot;\n      }\n    }\n  }\n}\nnode {\n  name: &q
uot;gradients/predictions_grad/MatMul&quot;\n  op: &quot;MatMul&
quot;\n  input: &quot;gradients/sub_grad/tuple/control_dependenc
y&quot;\n  input: &quot;theta/read&quot;\n  attr {\n    key: &qu
ot;T&quot;\n    value {\n      type: DT_FLOAT\n    }\n  }\n  att
r {\n    key: &quot;transpose_a&quot;\n    value {\n      b: fal
se\n    }\n  }\n  attr {\n    key: &quot;transpose_b&quot;\n
value {\n      b: true\n    }\n  }\n}\nnode {\n  name: &quot;gra
dients/predictions_grad/MatMul_1&quot;\n  op: &quot;MatMul&quot;
\n  input: &quot;X&quot;\n  input: &quot;gradients/sub_grad/tupl
e/control_dependency&quot;\n  attr {\n    key: &quot;T&quot;\n
  value {\n      type: DT_FLOAT\n    }\n  }\n  attr {\n    key:
&quot;transpose_a&quot;\n    value {\n      b: true\n    }\n  }\
n  attr {\n    key: &quot;transpose_b&quot;\n    value {\n
b: false\n    }\n  }\n}\nnode {\n  name: &quot;gradients/predict
ions_grad/tuple/group_deps&quot;\n  op: &quot;NoOp&quot;\n  inpu
t: &quot;^gradients/predictions_grad/MatMul&quot;\n  input: &quo
t;^gradients/predictions_grad/MatMul_1&quot;\n}\nnode {\n  name:
 &quot;gradients/predictions_grad/tuple/control_dependency&quot;
\n  op: &quot;Identity&quot;\n  input: &quot;gradients/predictio
ns_grad/MatMul&quot;\n  input: &quot;^gradients/predictions_grad
/tuple/group_deps&quot;\n  attr {\n    key: &quot;T&quot;\n    v
alue {\n      type: DT_FLOAT\n    }\n  }\n  attr {\n    key: &qu
```

```
ot;_class&quot;\n    value {\n      list {\n        s: &quot;loc
:@gradients/predictions_grad/MatMul&quot;\n      }\n    }\n  }\n
}\nnode {\n  name: &quot;gradients/predictions_grad/tuple/contro
l_dependency_1&quot;\n  op: &quot;Identity&quot;\n  input: &quot
;gradients/predictions_grad/MatMul_1&quot;\n  input: &quot;^grad
ients/predictions_grad/tuple/group_deps&quot;\n  attr {\n    key
: &quot;T&quot;\n    value {\n      type: DT_FLOAT\n    }\n  }\n
  attr {\n    key: &quot;_class&quot;\n    value {\n      list {
\n        s: &quot;loc:@gradients/predictions_grad/MatMul_1&quot
;\n      }\n    }\n  }\n}\nnode {\n  name: &quot;GradientDescent
/learning_rate&quot;\n  op: &quot;Const&quot;\n  attr {\n    key
: &quot;dtype&quot;\n    value {\n      type: DT_FLOAT\n    }\n
 }\n  attr {\n    key: &quot;value&quot;\n    value {\n      ten
sor {\n        dtype: DT_FLOAT\n        tensor_shape {\n
}\n        float_val: 0.009999999776482582\n      }\n    }\n  }\
n}\nnode {\n  name: &quot;GradientDescent/update_theta/ApplyGrad
ientDescent&quot;\n  op: &quot;ApplyGradientDescent&quot;\n  inp
ut: &quot;theta&quot;\n  input: &quot;GradientDescent/learning_r
ate&quot;\n  input: &quot;gradients/predictions_grad/tuple/contr
ol_dependency_1&quot;\n  attr {\n    key: &quot;T&quot;\n    val
ue {\n      type: DT_FLOAT\n    }\n  }\n  attr {\n    key: &quot
;_class&quot;\n    value {\n      list {\n        s: &quot;loc:@
theta&quot;\n      }\n    }\n  }\n  attr {\n    key: &quot;use_l
ocking&quot;\n    value {\n      b: false\n    }\n  }\n}\nnode {
\n  name: &quot;GradientDescent&quot;\n  op: &quot;NoOp&quot;\n
 input: &quot;^GradientDescent/update_theta/ApplyGradientDescent
&quot;\n}\nnode {\n  name: &quot;init&quot;\n  op: &quot;NoOp&qu
ot;\n  input: &quot;^theta/Assign&quot;\n}\nnode {\n  name: &quo
t;save/Const&quot;\n  op: &quot;Const&quot;\n  attr {\n    key:
&quot;dtype&quot;\n    value {\n      type: DT_STRING\n    }\n
}\n  attr {\n    key: &quot;value&quot;\n    value {\n      tens
or {\n        dtype: DT_STRING\n        tensor_shape {\n
}\n        string_val: &quot;model&quot;\n      }\n    }\n  }\n}
\nnode {\n  name: &quot;save/SaveV2/tensor_names&quot;\n  op: &q
uot;Const&quot;\n  attr {\n    key: &quot;dtype&quot;\n    value
 {\n      type: DT_STRING\n    }\n  }\n  attr {\n    key: &quot;
value&quot;\n    value {\n      tensor {\n        dtype: DT_STRI
NG\n        tensor_shape {\n          dim {\n            size: 1
\n          }\n        }\n        string_val: &quot;theta&quot;\
n      }\n    }\n  }\n}\nnode {\n  name: &quot;save/SaveV2/shape
```

```
_and_slices&quot;\n  op: &quot;Const&quot;\n  attr {\n    key: &
quot;dtype&quot;\n    value {\n      type: DT_STRING\n    }\n  }
\n  attr {\n    key: &quot;value&quot;\n    value {\n      tenso
r {\n        dtype: DT_STRING\n        tensor_shape {\n
 dim {\n            size: 1\n          }\n        }\n        str
ing_val: &quot;&quot;\n      }\n    }\n  }\n}\nnode {\n  name: &
quot;save/SaveV2&quot;\n  op: &quot;SaveV2&quot;\n  input: &quot
;save/Const&quot;\n  input: &quot;save/SaveV2/tensor_names&quot;
\n  input: &quot;save/SaveV2/shape_and_slices&quot;\n  input: &q
uot;theta&quot;\n  attr {\n    key: &quot;dtypes&quot;\n    valu
e {\n      list {\n        type: DT_FLOAT\n      }\n    }\n  }\n
}\nnode {\n  name: &quot;save/control_dependency&quot;\n  op: &q
uot;Identity&quot;\n  input: &quot;save/Const&quot;\n  input: &q
uot;^save/SaveV2&quot;\n  attr {\n    key: &quot;T&quot;\n    va
lue {\n      type: DT_STRING\n    }\n  }\n  attr {\n    key: &qu
ot;_class&quot;\n    value {\n      list {\n        s: &quot;loc
:@save/Const&quot;\n      }\n    }\n  }\n}\nnode {\n  name: &quo
t;save/RestoreV2/tensor_names&quot;\n  op: &quot;Const&quot;\n
attr {\n    key: &quot;dtype&quot;\n    value {\n      type: DT_
STRING\n    }\n  }\n  attr {\n    key: &quot;value&quot;\n    va
lue {\n      tensor {\n        dtype: DT_STRING\n        tensor_
shape {\n          dim {\n            size: 1\n          }\n
    }\n        string_val: &quot;theta&quot;\n      }\n    }\n
}\n}\nnode {\n  name: &quot;save/RestoreV2/shape_and_slices&quot
;\n  op: &quot;Const&quot;\n  attr {\n    key: &quot;dtype&quot;
\n    value {\n      type: DT_STRING\n    }\n  }\n  attr {\n
key: &quot;value&quot;\n    value {\n      tensor {\n        dty
pe: DT_STRING\n        tensor_shape {\n          dim {\n
    size: 1\n          }\n        }\n        string_val: &quot;&
quot;\n      }\n    }\n  }\n}\nnode {\n  name: &quot;save/Restor
eV2&quot;\n  op: &quot;RestoreV2&quot;\n  input: &quot;save/Cons
t&quot;\n  input: &quot;save/RestoreV2/tensor_names&quot;\n  inp
ut: &quot;save/RestoreV2/shape_and_slices&quot;\n  attr {\n    k
ey: &quot;dtypes&quot;\n    value {\n      list {\n        type:
 DT_FLOAT\n      }\n    }\n  }\n}\nnode {\n  name: &quot;save/As
sign&quot;\n  op: &quot;Assign&quot;\n  input: &quot;theta&quot;
\n  input: &quot;save/RestoreV2&quot;\n  attr {\n    key: &quot;
T&quot;\n    value {\n      type: DT_FLOAT\n    }\n  }\n  attr {
\n    key: &quot;_class&quot;\n    value {\n      list {\n
  s: &quot;loc:@theta&quot;\n      }\n    }\n  }\n  attr {\n
```

```
key: &quot;use_locking&quot;\n    value {\n      b: true\n    }\
n  }\n  attr {\n    key: &quot;validate_shape&quot;\n    value {
\n      b: true\n    }\n  }\n}\nnode {\n  name: &quot;save/resto
re_all&quot;\n  op: &quot;NoOp&quot;\n  input: &quot;^save/Assig
n&quot;\n}\n';
      }
    </script>
    <link rel=&quot;import&quot; href=&quot;https://tensorboard.
appspot.com/tf-graph-basic.build.html&quot; onload=load()>
    <div style=&quot;height:600px&quot;>
      <tf-graph-basic id=&quot;graph0.1784179106002547&quot;></t
f-graph-basic>
    </div>
"></iframe>
```

Visualization - using TensorBoard

- Start **TensorBoard**: $ tensorboard --logdir tf_logs/ (starts on localhost:6006)

```
tf.reset_default_graph()

# Need a logging directory for TB data
# with timestamps to avoid mixing runs together

from datetime import datetime
now         = datetime.utcnow().strftime("%Y%m%d%H%M%S")

root_logdir = "tf_logs"
logdir      = "{}/run-{}/".format(root_logdir, now)

n_epochs = 1000
learning_rate = 0.01

X = tf.placeholder(
    tf.float32,
    shape=(None, n + 1),
    name="X")

y = tf.placeholder(
```

```
    tf.float32,
    shape=(None, 1),
    name="y")

theta = tf.Variable(
    tf.random_uniform([n + 1, 1], -1.0, 1.0, seed=42),
    name="theta")

y_pred = tf.matmul(
    X, theta, name="predictions")

error = y_pred - y

mse = tf.reduce_mean(
    tf.square(error),
    name="mse")

optimizer = tf.train.GradientDescentOptimizer(
    learning_rate=learning_rate)

training_op = optimizer.minimize(mse)

init = tf.global_variables_initializer()

mse_summary = tf.summary.scalar('MSE', mse)

# FileWriter - creates logdir if not already present,
# then writes graph def to a binary logfile.

summary_writer = tf.summary.FileWriter(
    logdir,
    tf.get_default_graph())
```

```
n_epochs = 10
batch_size = 100
n_batches = int(np.ceil(m / batch_size))

with tf.Session() as sess:
    sess.run(init)
```

```
    for epoch in range(n_epochs):
        for batch_index in range(n_batches):

            X_batch, y_batch = fetch_batch(
                epoch,
                batch_index,
                batch_size)

            # evaluate mse_summary on periodic basis,
            # eg every 10 minibatches.
            # adds summary for addition to events file.

            if batch_index % 10 == 0:
                summary_str = mse_summary.eval(
                    feed_dict={X: X_batch, y: y_batch})

                step = epoch * n_batches + batch_index

                summary_writer.add_summary(
                    summary_str,
                    step)

            sess.run(
                training_op,
                feed_dict={X: X_batch, y: y_batch})

    best_theta = theta.eval()

summary_writer.flush()
summary_writer.close()
print("Best theta:")
print(best_theta)
```

```
Best theta:
[[ 2.07001591]
 [ 0.82045609]
 [ 0.1173173 ]
 [-0.22739051]
 [ 0.31134021]
 [ 0.00353193]
 [-0.01126994]
 [-0.91643935]
 [-0.87950081]]
```

```
#!ls tf_logs/run*
```

Name Scopes

- Graphs can contain thousands of nodes. *name scopes* group related nodes to aid visualization.

```
tf.reset_default_graph()

now = datetime.utcnow().strftime("%Y%m%d%H%M%S")
root_logdir = "tf_logs"
logdir = "{}/run-{}/".format(root_logdir, now)

n_epochs = 1000
learning_rate = 0.01

X = tf.placeholder(
    tf.float32,
    shape=(None, n + 1),
    name="X")

y = tf.placeholder(
    tf.float32,
    shape=(None, 1),
    name="y")
```

```
theta = tf.Variable(
    tf.random_uniform([n + 1, 1], -1.0, 1.0, seed=42),
    name="theta")

y_pred = tf.matmul(
    X, theta,
    name="predictions")

##### Name Scope
with tf.name_scope('loss') as scope:
    error = y_pred - y
    mse = tf.reduce_mean(tf.square(error), name="mse")
#####

optimizer = tf.train.GradientDescentOptimizer(
    learning_rate=learning_rate)

training_op = optimizer.minimize(mse)

init = tf.global_variables_initializer()

mse_summary = tf.summary.scalar(
    'MSE', mse)

summary_writer = tf.summary.FileWriter(
    logdir, tf.get_default_graph())
```

```
n_epochs = 10
batch_size = 100
n_batches = int(np.ceil(m / batch_size))

with tf.Session() as sess:
    sess.run(init)

    for epoch in range(n_epochs):
        for batch_index in range(n_batches):

            X_batch, y_batch = fetch_batch(
                epoch,
                batch_index,
                batch_size)

            if batch_index % 10 == 0:

                summary_str = mse_summary.eval(
                    feed_dict={X: X_batch, y: y_batch})

                step = epoch * n_batches + batch_index

                summary_writer.add_summary(
                    summary_str,
                    step)

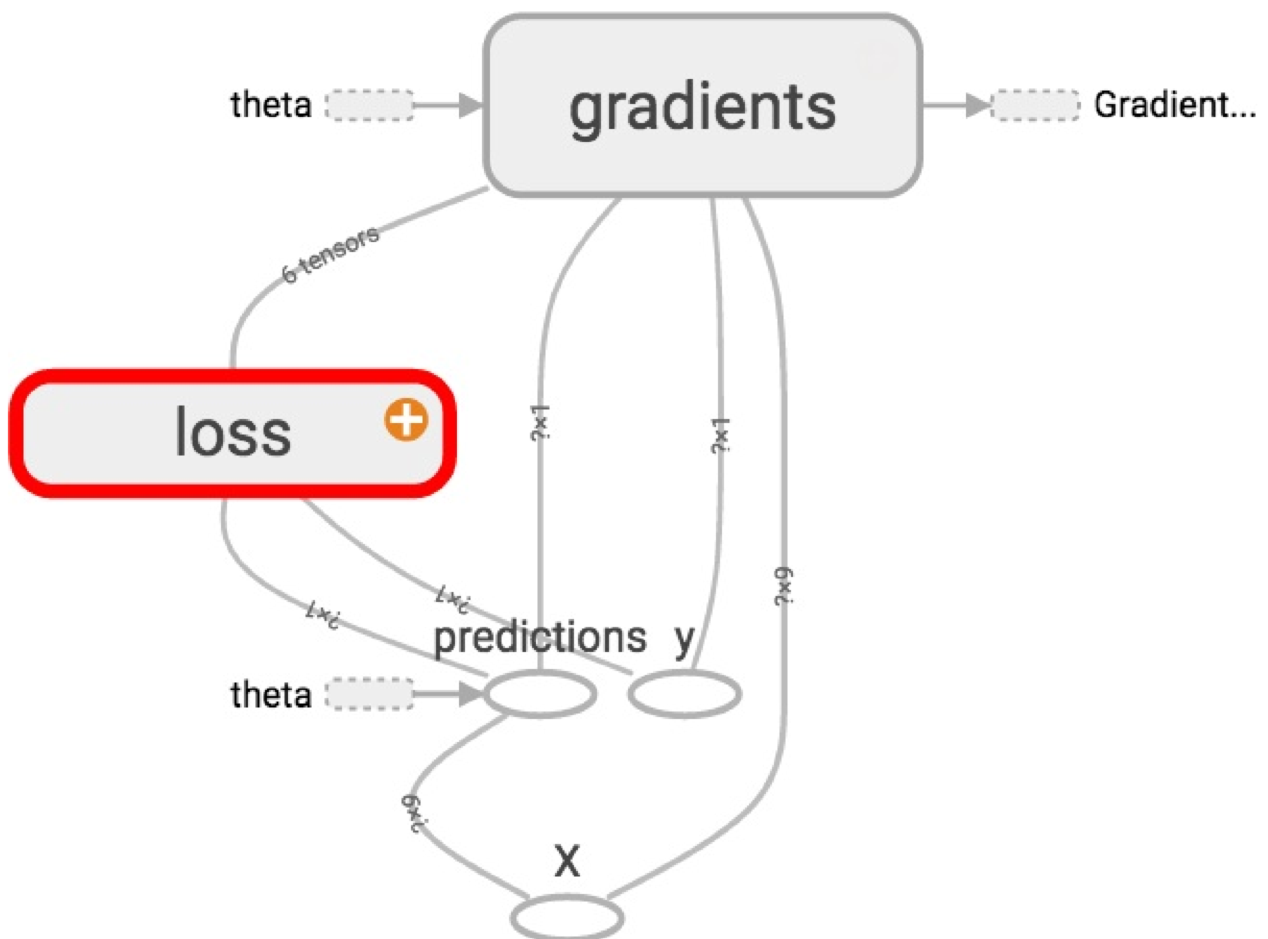
            sess.run(
                training_op,
                feed_dict={X: X_batch, y: y_batch})

    best_theta = theta.eval()

summary_writer.flush()
summary_writer.close()
print("Best theta:")
print(best_theta)
```

```
Best theta:
[[ 2.07001591]
 [ 0.82045609]
 [ 0.1173173 ]
 [-0.22739051]
 [ 0.31134021]
 [ 0.00353193]
 [-0.01126994]
 [-0.91643935]
 [-0.87950081]]
```

In TensorBoard:



Modularity

- ex: create graph, adds two ReLU nodes
- output: result if >0, 0 otherwise

```
# UGLY

tf.reset_default_graph()

n_features = 3
X = tf.placeholder(
    tf.float32,
    shape=(None, n_features),
    name="X")

w1 = tf.Variable(
    tf.random_normal(
        (n_features, 1)),
    name="weights1")

w2 = tf.Variable(
    tf.random_normal(
        (n_features, 1)),
    name="weights2")

b1 = tf.Variable(
    0.0, name="bias1")
b2 = tf.Variable(
    0.0, name="bias2")

linear1 = tf.add(
    tf.matmul(X, w1), b1, name="linear1")

linear2 = tf.add(
    tf.matmul(X, w2), b2, name="linear2")

relu1 = tf.maximum(
    linear1, 0, name="relu1")

relu2 = tf.maximum(
    linear2, 0, name="relu2")

output = tf.add_n([relu1, relu2], name="output")
```

```
# better -- you can create functions that build ReLUs!

tf.reset_default_graph()

def relu(X):
    w_shape = int(
        X.get_shape()[1]), 1

    w = tf.Variable(
        tf.random_normal(w_shape),
        name="weights")

    b = tf.Variable(
        0.0,
        name="bias")

    linear = tf.add(
        tf.matmul(X, w),
        b,
        name="linear")

    return tf.maximum(linear, 0, name="relu")

n_features = 3

X = tf.placeholder(
    tf.float32,
    shape=(None, n_features),
    name="X")

relus = [relu(X) for i in range(5)]

output = tf.add_n(
    relus,
    name="output")

summary_writer = tf.summary.FileWriter(
    "logs/relu1",
    tf.get_default_graph())
```

Sharing Variables

- Simplest option: define it first, then share it as parameter to all functions that need it.

```
# better, with name scopes

tf.reset_default_graph()

def relu(X):
    with tf.name_scope("relu"):

        w_shape = int(
            X.get_shape()[1]), 1

        w = tf.Variable(
            tf.random_normal(w_shape), name="weights")

        b = tf.Variable(
            0.0, name="bias")

        linear = tf.add(
            tf.matmul(X, w), b, name="linear")

        return tf.maximum(
            linear, 0, name="max")

n_features = 3

X = tf.placeholder(
    tf.float32,
    shape=(None, n_features),
    name="X")

relus = [relu(X) for i in range(5)]

output = tf.add_n(
    relus, name="output")

summary_writer = tf.summary.FileWriter(
    "logs/relu2",
    tf.get_default_graph())

summary_writer.close()
```

```
!ls logs
```

```
relu1  relu2  relu6
```

```
tf.reset_default_graph()

def relu(X, threshold):
    with tf.name_scope("relu"):
        w_shape = int(X.get_shape()[1]), 1
        w = tf.Variable(tf.random_normal(w_shape), name="weights"
)
        b = tf.Variable(0.0, name="bias")
        linear = tf.add(tf.matmul(X, w), b, name="linear")
        return tf.maximum(linear, threshold, name="max")

threshold = tf.Variable(0.0, name="threshold")
X = tf.placeholder(tf.float32, shape=(None, n_features), name="X"
)
relus = [relu(X, threshold) for i in range(5)]
output = tf.add_n(relus, name="output")
```

```
tf.reset_default_graph()

def relu(X):
    with tf.name_scope("relu"):
        if not hasattr(relu, "threshold"):
            relu.threshold = tf.Variable(0.0, name="threshold")
        w_shape = int(X.get_shape()[1]), 1
        w = tf.Variable(tf.random_normal(w_shape), name="weights"
)
        b = tf.Variable(0.0, name="bias")
        linear = tf.add(tf.matmul(X, w), b, name="linear")
        return tf.maximum(linear, relu.threshold, name="max")

X = tf.placeholder(tf.float32, shape=(None, n_features), name="X"
)
relus = [relu(X) for i in range(5)]
output = tf.add_n(relus, name="output")
```

```
tf.reset_default_graph()

def relu(X):
    with tf.variable_scope("relu", reuse=True):
        threshold = tf.get_variable("threshold", shape=(), initi
alizer=tf.constant_initializer(0.0))
        w_shape = int(X.get_shape()[1]), 1
        w = tf.Variable(tf.random_normal(w_shape), name="weights"
)
        b = tf.Variable(0.0, name="bias")
        linear = tf.add(tf.matmul(X, w), b, name="linear")
        return tf.maximum(linear, threshold, name="max")

X = tf.placeholder(tf.float32, shape=(None, n_features), name="X"
)
with tf.variable_scope("relu"):
    threshold = tf.get_variable("threshold", shape=(), initializ
er=tf.constant_initializer(0.0))
relus = [relu(X) for i in range(5)]
output = tf.add_n(relus, name="output")

summary_writer = tf.summary.FileWriter("logs/relu6", tf.get_defa
ult_graph())
summary_writer.close()
```

Intro - Perceptrons

- Simplest ANN architecture
- Uses *linear threshold unit (LTU)* - returns weight sum of inputs, applies *step function* to sum, outputs result
- Single LTU can be used for simple linear binary classification
- Perceptron = single layer of LTUs, each one connected to all inputs
- Percepton training based on *Hebb's Rule*. (basically, connection weight between two neurons goes up when they have same output.)
- Linear decision boundary, so Perceptrons not capable of learning complex patterns.

```
# Perceptron with Iris dataset (Scikit)

import numpy as np

from sklearn.datasets import load_iris
iris = load_iris()

X = iris.data[:, (2, 3)]  # petal length, petal width
y = (iris.target == 0).astype(np.int)
```

```
from sklearn.linear_model import Perceptron

per_clf = Perceptron(random_state=42)
per_clf.fit(X, y)

y_pred = per_clf.predict([[2, 0.5]])
print(y_pred)
```

```
[1]
```

- Perceptron learning algo very similar go SGD.
- Perceptrons do provide class probability (like Logistic Regression classsifier).

They simply make predictions based on hard threshold.
- Some limitations can be eliminated with stacked Perceptrons.

```
import matplotlib.pyplot as plt

a = -per_clf.coef_[0][0] / per_clf.coef_[0][1]
b = -per_clf.intercept_ / per_clf.coef_[0][1]

axes = [0, 5, 0, 2]

x0, x1 = np.meshgrid(
        np.linspace(axes[0], axes[1], 500).reshape(-1, 1),
        np.linspace(axes[2], axes[3], 200).reshape(-1, 1),
    )

X_new = np.c_[
    x0.ravel(),
    x1.ravel()]

y_predict = per_clf.predict(X_new)

zz = y_predict.reshape(x0.shape)

plt.figure(figsize=(10, 4))
plt.plot(X[y==0, 0], X[y==0, 1], "bs", label="Not Iris-Setosa")
plt.plot(X[y==1, 0], X[y==1, 1], "yo", label="Iris-Setosa")

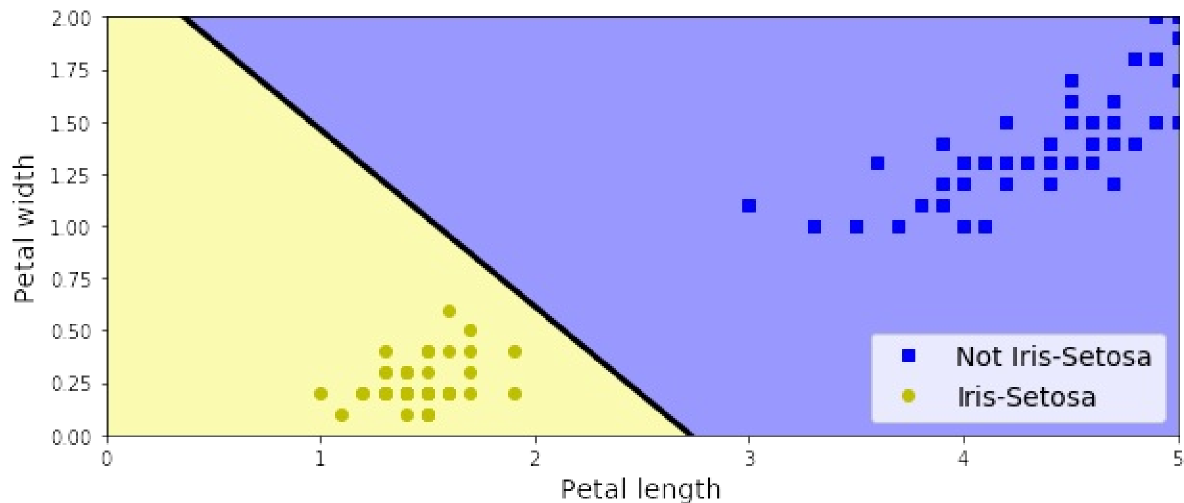
plt.plot(
    [axes[0],
     axes[1]],
    [a * axes[0] + b,
     a * axes[1] + b],
    "k-", linewidth=3)

from matplotlib.colors import ListedColormap
custom_cmap = ListedColormap(['#9898ff', '#fafab0'])

plt.contourf(x0, x1, zz, cmap=custom_cmap, linewidth=5)
plt.xlabel("Petal length", fontsize=14)
plt.ylabel("Petal width", fontsize=14)
```

```
plt.legend(loc="lower right", fontsize=14)
plt.axis(axes)

#save_fig("perceptron_iris_plot")
plt.show()
```



MLPs and Backpropagation

- MLP contains one input layer, at least one hidden layer (LTU based), and one output layer (LTU based).
- Backpropagation intro'd in 1986 paper. Can be described as Gradient Descent using reverse-mode autodiff.
- For each training instance:
    - Find output of each node in each consecutive layer (forward pass).
    - Measure output error & how much each node in last hidden layer contributed to it.
    - Measure how much each node in *previous* hidden layer contributed to *this* hidden layer.
    - Repeat until *input* layer is reached (backward pass).
    - Adjust each connection weight to reduce the error.
- To make algorithm work, MLP architecture changed to use logistic function delta(z) = 1/(1+exp(-z)) instead of step function.
- Backpropagation can also use hyperbolic tangent or ReLU functions if desired.

```
# Activation functions

def logit(z):
    return 1 / (1 + np.exp(-z))

def relu(z):
    return np.maximum(0, z)

def derivative(f, z, eps=0.000001):
    return (f(z + eps) - f(z - eps))/(2 * eps)
```

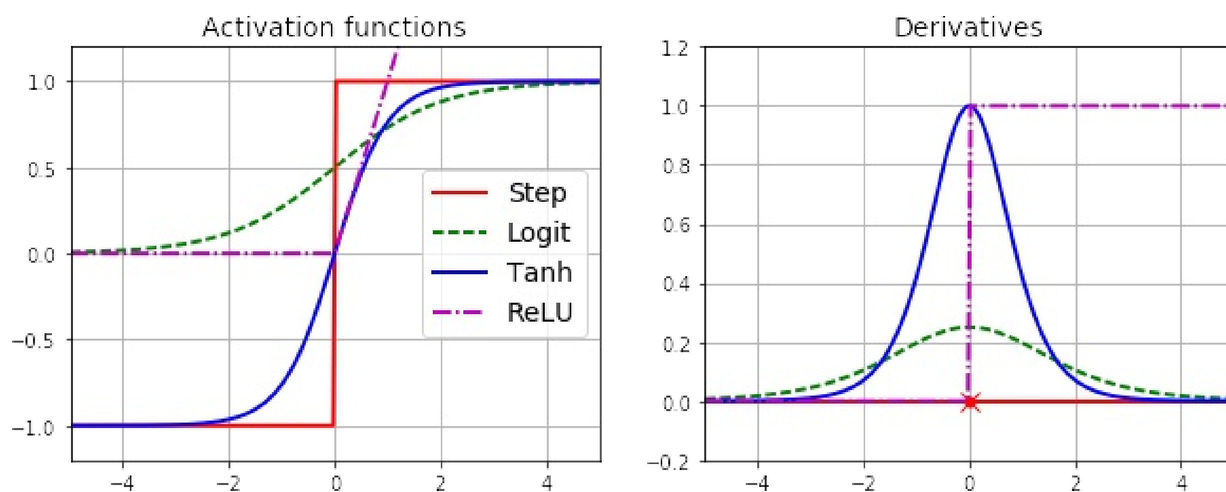
```
z = np.linspace(-5, 5, 200)

plt.figure(figsize=(11,4))

plt.subplot(121)
plt.plot(z, np.sign(z), "r-", linewidth=2, label="Step")
plt.plot(z, logit(z), "g--", linewidth=2, label="Logit")
plt.plot(z, np.tanh(z), "b-", linewidth=2, label="Tanh")
plt.plot(z, relu(z), "m-.", linewidth=2, label="ReLU")
plt.grid(True)
plt.legend(loc="center right", fontsize=14)
plt.title("Activation functions", fontsize=14)
plt.axis([-5, 5, -1.2, 1.2])

plt.subplot(122)
plt.plot(z, derivative(np.sign, z), "r-", linewidth=2, label="St
ep")
plt.plot(0, 0, "ro", markersize=5)
plt.plot(0, 0, "rx", markersize=10)
plt.plot(z, derivative(logit, z), "g--", linewidth=2, label="Log
it")
plt.plot(z, derivative(np.tanh, z), "b-", linewidth=2, label="Ta
nh")
plt.plot(z, derivative(relu, z), "m-.", linewidth=2, label="ReLU"
)
plt.grid(True)
#plt.legend(loc="center right", fontsize=14)
plt.title("Derivatives", fontsize=14)
plt.axis([-5, 5, -0.2, 1.2])

#save_fig("activation_functions_plot")
plt.show()
```




```
# activation functions, continued
def heaviside(z):
    return (z >= 0).astype(z.dtype)

def sigmoid(z):
    return 1/(1+np.exp(-z))

def mlp_xor(x1, x2, activation=heaviside):
    return activation(
        -activation(x1 + x2 - 1.5) + activation(x1 + x2 - 0.5) -
0.5)
```

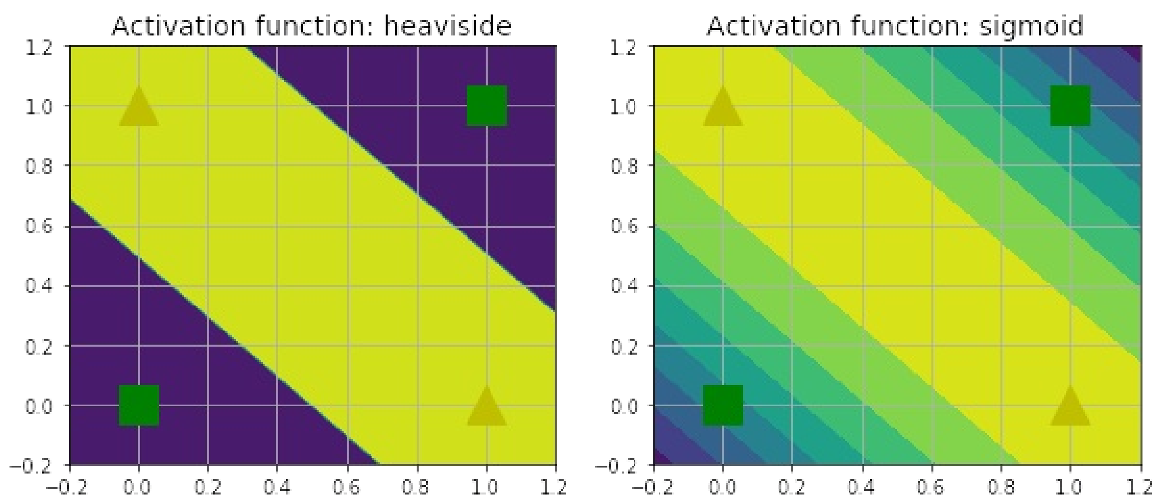
```
x1s = np.linspace(-0.2, 1.2, 100)
x2s = np.linspace(-0.2, 1.2, 100)
x1, x2 = np.meshgrid(x1s, x2s)

z1 = mlp_xor(x1, x2, activation=heaviside)
z2 = mlp_xor(x1, x2, activation=sigmoid)

plt.figure(figsize=(10,4))

plt.subplot(121)
plt.contourf(x1, x2, z1)
plt.plot([0, 1], [0, 1], "gs", markersize=20)
plt.plot([0, 1], [1, 0], "y^", markersize=20)
plt.title("Activation function: heaviside", fontsize=14)
plt.grid(True)

plt.subplot(122)
plt.contourf(x1, x2, z2)
plt.plot([0, 1], [0, 1], "gs", markersize=20)
plt.plot([0, 1], [1, 0], "y^", markersize=20)
plt.title("Activation function: sigmoid", fontsize=14)
plt.grid(True)
plt.show()
```



MLP Training

- MLP often used for classification - each output corresponding to distinct

binary class (ex: urgent/not-urgent, spam/not-spam, ...)
- If exclusive classes, output layer often uses shared **softmax** function.

DNN Training with "plain" TF

- Use **mini-batch gradient descent** on MNIST dataset
- Specify #inputs, #outputs, #hidden neurons in each layer

```
import tensorflow as tf

tf.reset_default_graph()

n_inputs = 28*28  # MNIST
n_hidden1 = 300
n_hidden2 = 100
n_outputs = 10
learning_rate = 0.01

# placeholders for training data & targets
# X,y only partially defined due to unknown #instances in traini
ng batches

X = tf.placeholder(tf.float32, shape=(None, n_inputs), name="X")
y = tf.placeholder(tf.int64,   shape=(None),           name="y")
```

```
# now need to create two hidden layers + one output layer

'''
No need to define your own. TF shortcuts:
fully_connected()
'''

def neuron_layer(X, n_neurons, name, activation=None):

    # define a name scope to aid readability
    with tf.name_scope(name):

        n_inputs = int(X.get_shape()[1])

        # create weights matrix. 2D (#inputs, #neurons)
        # randomly initialized w/ truncated Gaussian, stdev = 2/
sqrt(#inputs)
        # aids convergence speed

        stddev = 1 / np.sqrt(n_inputs)
        init = tf.truncated_normal((n_inputs, n_neurons), stddev
=stddev)
        W = tf.Variable(init, name="weights")

        # create bias variable, initialized to zero, one param p
er neuron
        b = tf.Variable(tf.zeros([n_neurons]), name="biases")

        # Z = X dot W + b
        Z = tf.matmul(X, W) + b

        # return relu(z), or simply z
        if activation=="relu":
            return tf.nn.relu(Z)
        else:
            return Z
```

```
with tf.name_scope("dnn"):
```

```
    hidden1 = neuron_layer(X,       n_hidden1, "hidden1", activa
tion="relu")
    hidden2 = neuron_layer(hidden1, n_hidden2, "hidden2", activa
tion="relu")

    # logits = NN output before going thru softmax activation
    logits =  neuron_layer(hidden2, n_outputs, "output")

with tf.name_scope("loss"):

    # sparse_softmax_cross_entropy_with_logits() -- TF routine,
handles corner cases for you.
    xentropy = tf.nn.sparse_softmax_cross_entropy_with_logits(
        labels=y,
        logits=logits)

    # use reduce_mean() to find mean cross-entropy over all inst
ances.
    loss = tf.reduce_mean(
        xentropy,
        name="loss")

# use GD to handle cost function, ie minimize loss
with tf.name_scope("train"):
    optimizer = tf.train.GradientDescentOptimizer(learning_rate)
    training_op = optimizer.minimize(loss)

# use accuracy as performance measure.

with tf.name_scope("eval"):       # verify whether highest logi
t corresponds to target class
    correct = tf.nn.in_top_k(      # using in_top_k(), returns 1
D tensor of booleans
        logits, y, 1)

    accuracy = tf.reduce_mean(             # recast booleans to
float & find avg.
        tf.cast(correct, tf.float32))      # this gives overall
accuracy number.
```

```
init = tf.global_variables_initializer()    # initializer node
saver = tf.train.Saver()                    # to save trained par
ams to disk
```

Execution Phase

- Load MNIST using TF helpers (fetch, auto-scale, shuffle, provide minibatch function)

```
# load MNIST

from tensorflow.examples.tutorials.mnist import input_data
mnist = input_data.read_data_sets("/tmp/data/")

n_epochs = 20
batch_size = 50
```

```
Extracting /tmp/data/train-images-idx3-ubyte.gz
Extracting /tmp/data/train-labels-idx1-ubyte.gz
Extracting /tmp/data/t10k-images-idx3-ubyte.gz
Extracting /tmp/data/t10k-labels-idx1-ubyte.gz
```

```
# Train
with tf.Session() as sess:

    init.run() # initialize all variables

    for epoch in range(n_epochs):
        for iteration in range(mnist.train.num_examples // batch
_size):

            # use next_batch() to fetch data
            X_batch, y_batch = mnist.train.next_batch(batch_size
)

            sess.run(
                training_op,
                feed_dict={X: X_batch, y: y_batch})

        acc_train = accuracy.eval(
            feed_dict={X: X_batch, y: y_batch})

        acc_test = accuracy.eval(
            feed_dict={X: mnist.test.images,
                       y: mnist.test.labels})

        print(epoch, "Train accuracy:", acc_train, "Test accurac
y:", acc_test)

        save_path = saver.save(sess, "./my_model_final.ckpt")
```

```
0 Train accuracy: 0.94 Test accuracy: 0.8753
1 Train accuracy: 0.9 Test accuracy: 0.9087
2 Train accuracy: 0.94 Test accuracy: 0.9212
3 Train accuracy: 0.88 Test accuracy: 0.9239
4 Train accuracy: 0.88 Test accuracy: 0.9335
5 Train accuracy: 0.96 Test accuracy: 0.9365
6 Train accuracy: 0.92 Test accuracy: 0.9415
7 Train accuracy: 0.94 Test accuracy: 0.9449
8 Train accuracy: 0.96 Test accuracy: 0.9459
9 Train accuracy: 0.94 Test accuracy: 0.9497
10 Train accuracy: 1.0 Test accuracy: 0.9539
11 Train accuracy: 0.94 Test accuracy: 0.9563
12 Train accuracy: 0.98 Test accuracy: 0.958
13 Train accuracy: 0.98 Test accuracy: 0.9584
14 Train accuracy: 0.94 Test accuracy: 0.9614
15 Train accuracy: 1.0 Test accuracy: 0.9622
16 Train accuracy: 0.96 Test accuracy: 0.9639
17 Train accuracy: 0.92 Test accuracy: 0.9635
18 Train accuracy: 0.98 Test accuracy: 0.9654
19 Train accuracy: 0.92 Test accuracy: 0.9667
```

Using in production

- Now trained - you can use the NN to predict.

```
with tf.Session() as sess:

    # load from disk
    saver.restore(sess, save_path) #"my_model_final.ckpt")

    # grab images you want to classify
    X_new_scaled = mnist.test.images[:20]

    Z = logits.eval(feed_dict={X: X_new_scaled})
    print(np.argmax(Z, axis=1))
    print(mnist.test.labels[:20])
```

```
[7 2 1 0 4 1 4 9 6 9 0 6 9 0 1 5 9 7 3 4]
[7 2 1 0 4 1 4 9 5 9 0 6 9 0 1 5 9 7 3 4]
```

Parameter Tuning

- Way too many parameters - Grid Search approach not time-effective.
- 1st option: randomized search.
- 2nd option: Oscar
- Start with common defaults to restrict search space.

Number of hidden layers

- Deep nets have **much better parameter efficiency** than shallow ones. (They can model complex functions with much fewer neurons.)
- Largely due to hierarchical nature of most data modeling probs

Number of neurons per hidden layer

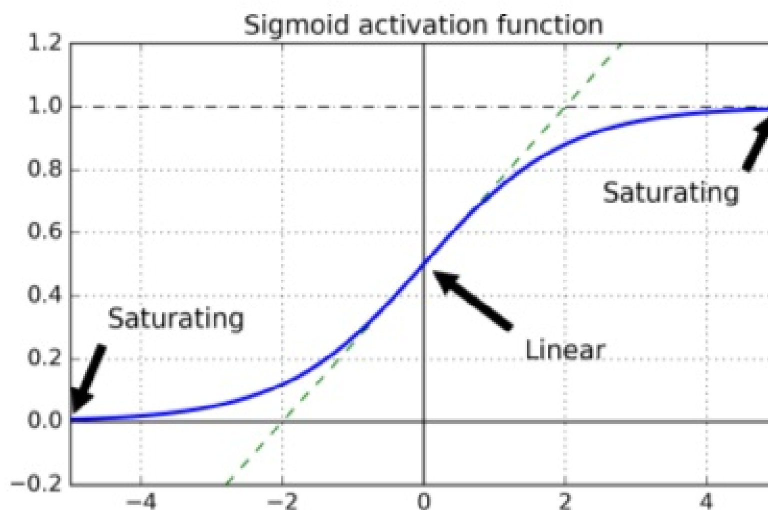
- Determined by input dimensions. Ex: MNIST requires 28x28 inputs, 10 outputs
- Try increasing # of layers before # neurons/layer.
- Simple trick: pick model w/ excessive layers & neurons, use early stopping, regularization, dropout, etc. to prevent overfit.

Activation functions

- Defaults:
  - use ReLU in hidden layers. Faster & helps avoid GD getting stuck on local plateaus.
  - use Softmax in output layer (for classification; none needed for regression.)

Vanishing & Exploding Gradients

- gradients get smaller as algorithm progresses to lower layers. Eventually GD leaves lower weights virtually unchanged. so training never converges.
- gradients can also grow out of control (often seen in RNNs).
- Significant paper - using combo of logistic sigmoid activiation with random weight initialization (normal, mean=0, stdev=1) -- output variance was >> input variance.
- logistic activation: function saturates at 0 or 1 with derivative very close to 0 ==> so backpropagation has no gradient to use.



Xavier & He Initialization

- For signals to flow properly in both directions, each layer's output variance should equal its input variance.
- Recommends initializing connection weights with random settings using #ins, #outs

Activation function	Uniform distribution [−r, r]	Normal distribution
Logistic	$r = \sqrt{\frac{6}{n_{\text{inputs}} + n_{\text{outputs}}}}$	$\sigma = \sqrt{\frac{2}{n_{\text{inputs}} + n_{\text{outputs}}}}$
Hyperbolic tangent	$r = 4\sqrt{\frac{6}{n_{\text{inputs}} + n_{\text{outputs}}}}$	$\sigma = 4\sqrt{\frac{2}{n_{\text{inputs}} + n_{\text{outputs}}}}$
ReLU (and its variants)	$r = \sqrt{2}\sqrt{\frac{6}{n_{\text{inputs}} + n_{\text{outputs}}}}$	$\sigma = \sqrt{2}\sqrt{\frac{2}{n_{\text{inputs}} + n_{\text{outputs}}}}$

- Default: *fully_connected()* function uses Xavier initialization w/ uniform distribution. Change to He initialization by using *variance_scaling_initializer()*

function

```
import tensorflow as tf
from tensorflow.contrib.layers import fully_connected

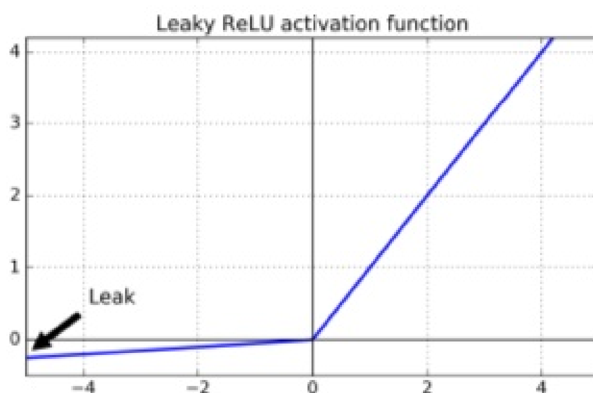
n_inputs = 28*28
n_hidden1 = 300

X = tf.placeholder(tf.float32, shape=(None, n_inputs), name="X")

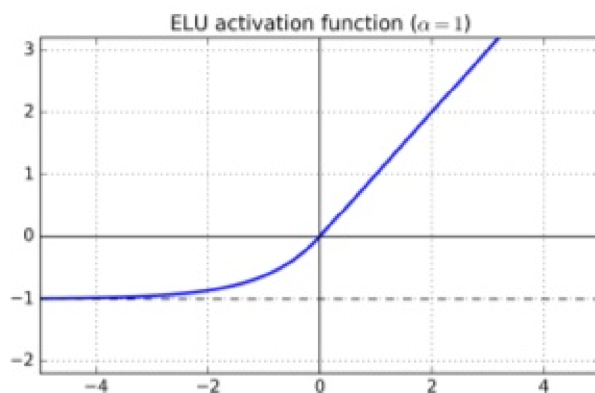
he_init = tf.contrib.layers.variance_scaling_initializer()
hidden1 = fully_connected(X, n_hidden1, weights_initializer=he_init, scope="h1")
```

Non-Saturating Activation Functions

- ReLU activations suffer from *dying ReLU* problem (they stop emitting anything other than zero).
- Workaround: the **leaku ReLU**. Alpha defines leakage; typical set to 0.01.



- Also: **randomized leaky ReLU (RReLU)** (randomized alpha)
- Also: **parametric leaky RuLE (PReLU)** (alpha can be modified during backprop)
- Also: **exponential linear unit (ELU)**. Allows negative values when z<0; non-zero gradient for z<0 (avoids dying units issue); smooth function everywhere. Uses exponential function, so harder to compute. paper




```
# TF doesn't have leaky ReLU predefined, but easy to build.

def leaky_relu(z, name=None):
    return tf.maximum(0.01 * z, z, name=name)

hidden1 = fully_connected(X, n_hidden1, activation_fn=leaky_relu
)
```

Batch Normalization

- proposed to solve vanishing/exploding gradients.
- Idea: pror to activation function, 1) zero-center & normalize inputs 2) scale & shift result with 2 new params per layer
- Net effect: model learns optimal scale & mean of inputs for each layer

1. $\mu_B = \frac{1}{m_B} \sum_{i=1}^{m_B} \mathbf{x}^{(i)}$
2. $\sigma_B{}^2 = \frac{1}{m_B} \sum_{i=1}^{m_B} \left(\mathbf{x}^{(i)} - \mu_B\right)^2$
3. $\hat{\mathbf{x}}^{(i)} = \frac{\mathbf{x}^{(i)} - \mu_B}{\sqrt{\sigma_B{}^2 + \epsilon}}$
4. $\mathbf{z}^{(i)} = \gamma \hat{\mathbf{x}}^{(i)} + \beta$

- Algorithm:
- Does add computational complexity. Consider plain ELU + He initializaton as well.

Batch Normalization with TF

- *batch_normalization()* - centers & normalizes inputs
- *batch_norm()* - above, plus finds mean, stdev, scaling, offset params
- call directly or include it in *fully_connected()* arguments

```
# use MNIST dataset again
from tensorflow.examples.tutorials.mnist import input_data
mnist = input_data.read_data_sets("/tmp/data/")
```

```
Successfully downloaded train-images-idx3-ubyte.gz 9912422 bytes
.
Extracting /tmp/data/train-images-idx3-ubyte.gz
Successfully downloaded train-labels-idx1-ubyte.gz 28881 bytes.
Extracting /tmp/data/train-labels-idx1-ubyte.gz
Successfully downloaded t10k-images-idx3-ubyte.gz 1648877 bytes.
Extracting /tmp/data/t10k-images-idx3-ubyte.gz
Successfully downloaded t10k-labels-idx1-ubyte.gz 4542 bytes.
Extracting /tmp/data/t10k-labels-idx1-ubyte.gz
```

```
#setup

import tensorflow as tf
from tensorflow.contrib.layers import batch_norm, fully_connecte
d

tf.reset_default_graph()

n_inputs = 28 * 28
n_hidden1 = 300
n_hidden2 = 100
n_outputs = 10
learning_rate = 0.01

X = tf.placeholder(tf.float32, shape=(None, n_inputs), name="X")
y = tf.placeholder(tf.int64, shape=(None), name="y")

def leaky_relu(z, name=None):
  return tf.maximum(0.01 * z, z, name=name)
```

```
# is_training: tells batch_norm() whether to use current minibat
ch's mean & stdev
# (found during training) or use running avgs (during testing)

with tf.name_scope("dnn"):
    hidden1 = fully_connected(X, n_hidden1, activation_fn=leaky_
relu, scope="hidden1")
    hidden2 = fully_connected(hidden1, n_hidden2, activation_fn=
leaky_relu, scope="hidden2")
    logits = fully_connected(hidden2, n_outputs, activation_fn=N
one, scope="outputs")

with tf.name_scope("loss"):
    xentropy = tf.nn.sparse_softmax_cross_entropy_with_logits(la
bels=y, logits=logits)
    loss = tf.reduce_mean(xentropy, name="loss")

with tf.name_scope("train"):
    optimizer = tf.train.GradientDescentOptimizer(learning_rate)
    training_op = optimizer.minimize(loss)

with tf.name_scope("eval"):
    correct = tf.nn.in_top_k(logits, y, 1)
    accuracy = tf.reduce_mean(tf.cast(correct, tf.float32))

init = tf.global_variables_initializer()
saver = tf.train.Saver()
```

```
n_epochs = 20
batch_size = 100

with tf.Session() as sess:

    init.run()

    for epoch in range(n_epochs):

        for iteration in range(len(mnist.test.labels)//batch_size):

            X_batch, y_batch = mnist.train.next_batch(batch_size)
            sess.run(training_op, feed_dict={X: X_batch, y: y_batch})

        acc_train = accuracy.eval(feed_dict={X: X_batch, y: y_batch})
        acc_test = accuracy.eval(feed_dict={X: mnist.test.images, y: mnist.test.labels})

        print(epoch, "Train accuracy:", acc_train, "Test accuracy:", acc_test)

    save_path = saver.save(sess, "my_model_final.ckpt")
```

```
0 Train accuracy: 0.6 Test accuracy: 0.642
1 Train accuracy: 0.73 Test accuracy: 0.7824
2 Train accuracy: 0.81 Test accuracy: 0.827
3 Train accuracy: 0.84 Test accuracy: 0.8539
4 Train accuracy: 0.8 Test accuracy: 0.8686
5 Train accuracy: 0.87 Test accuracy: 0.8759
6 Train accuracy: 0.85 Test accuracy: 0.8843
7 Train accuracy: 0.91 Test accuracy: 0.8903
8 Train accuracy: 0.86 Test accuracy: 0.8969
9 Train accuracy: 0.91 Test accuracy: 0.9018
10 Train accuracy: 0.91 Test accuracy: 0.9014
11 Train accuracy: 0.86 Test accuracy: 0.9065
12 Train accuracy: 0.88 Test accuracy: 0.9078
13 Train accuracy: 0.87 Test accuracy: 0.91
14 Train accuracy: 0.93 Test accuracy: 0.911
15 Train accuracy: 0.9 Test accuracy: 0.9123
16 Train accuracy: 0.91 Test accuracy: 0.9141
17 Train accuracy: 0.9 Test accuracy: 0.9149
18 Train accuracy: 0.92 Test accuracy: 0.9159
19 Train accuracy: 0.93 Test accuracy: 0.9174
```

Gradient Clipping

- to limit exploding gradients problem. Clip during backprop.
- Typical use case: recurrent NNs.
- source
- Uses TF *minimize()* function in optimizer.

```
threshold = 1.0

optimizer = tf.train.GradientDescentOptimizer(
    learning_rate)

grads_and_vars = optimizer.compute_gradients(
    loss)

capped_gvs = [
    (tf.clip_by_value(
        grad, -threshold, threshold), var)
    for grad, var in grads_and_vars]

training_op = optimizer.apply_gradients(capped_gvs)
```

Pretrained Layers & Reuse

- best practice: look for existing NN that tackles similar task, then reuse lower layers (aka *transfer learning*).

```
# Reuse with TF
```

Reusing Models from Other Frameworks

- Requires manual loading of weights (ex: Theano)
- Very tedious

```
'''
original_w = [] # Load the weights from the other framework
original_b = [] # Load the biases from the other framework

X = tf.placeholder(tf.float32, shape=(None, n_inputs), name="X")
hidden1 = fully_connected(X, n_hidden1, scope="hidden1")

[...] # # Build the rest of the model

# Get a handle on the variables created by fully_connected()
```

```
with tf.variable_scope("", default_name="", reuse=True): # root
scope
    hidden1_weights = tf.get_variable("hidden1/weights")
    hidden1_biases = tf.get_variable("hidden1/biases")

# Create nodes to assign arbitrary values to the weights and bia
ses
original_weights = tf.placeholder(tf.float32, shape=(n_inputs, n
_hidden1))
original_biases = tf.placeholder(tf.float32, shape=(n_hidden1))

assign_hidden1_weights = tf.assign(hidden1_weights, original_wei
ghts)
assign_hidden1_biases = tf.assign(hidden1_biases, original_biase
s)

init = tf.global_variables_initializer()

with tf.Session() as sess:
    sess.run(init)
    sess.run(
        assign_hidden1_weights,
        feed_dict={original_weights: original_w})

    sess.run(
        assign_hidden1_biases,
        feed_dict={original_biases: original_b})

    [...] # Train the model on your new task
'''
```

```
'\noriginal_w = [] # Load the weights from the other framework\n
original_b = [] # Load the biases from the other framework\n\nX
= tf.placeholder(tf.float32, shape=(None, n_inputs), name="X")\n
hidden1 = fully_connected(X, n_hidden1, scope="hidden1")\n\n[...
] # # Build the rest of the model\n\n# Get a handle on the varia
bles created by fully_connected()\n\nwith tf.variable_scope("",
default_name="", reuse=True): # root scope\n    hidden1_weights
= tf.get_variable("hidden1/weights")\n    hidden1_biases = tf.ge
t_variable("hidden1/biases")\n\n# Create nodes to assign arbitra
ry values to the weights and biases\noriginal_weights = tf.place
holder(tf.float32, shape=(n_inputs, n_hidden1))\noriginal_biases
 = tf.placeholder(tf.float32, shape=(n_hidden1))\n\nassign_hidde
n1_weights = tf.assign(hidden1_weights, original_weights)\nassig
n_hidden1_biases = tf.assign(hidden1_biases, original_biases)\n\
ninit = tf.global_variables_initializer()\n\nwith tf.Session() a
s sess:\n    sess.run(init)\n    sess.run(\n        assign_hidde
n1_weights, \n        feed_dict={original_weights: original_w})\
n        \n    sess.run(\n        assign_hidden1_biases, \n
   feed_dict={original_biases: original_b})\n        \n    [...]
 # Train the model on your new task\n'
```

Freezing Lower Layers

- If 1st DNN already learned low-level features, try to reuse them by freezing the weights.
- simplest way:

```
# provide all trainable var in hidden layers 3,4 & outputs to op
timizer function
# (this omits vars in hidden layers 1,2)
'''
train_vars = tf.get_collection(
    tf.GraphKeys.TRAINABLE_VARIABLES,
    scope="hidden[34]|outputs")

# minimizer can't touch layers 1,2 - they're "frozen"

training_op = optimizer.minimize(
    loss,
    var_list=train_vars)
'''
```

```
'\ntrain_vars = tf.get_collection(\n    tf.GraphKeys.TRAINABLE_V
ARIABLES,\n    scope="hidden[34]|outputs")\n\n# minimizer can\'t
 touch layers 1,2 - they\'re "frozen"\n\ntraining_op = optimizer
.minimize(\n    loss, \n    var_list=train_vars)\n'
```

Caching Lower Layers

- Huge speed boost!

```
'''import numpy as np

n_epochs = 100
n_batches = 500

for epoch in range(n_epochs):
    shuffled_idx = rnd.permutation(
        len(hidden2_outputs))

    hidden2_batches = np.array_split(
        hidden2_outputs[shuffled_idx],
        n_batches)

y_batches = np.array_split(
    y_train[shuffled_idx],
    n_batches)

for hidden2_batch, y_batch in zip(hidden2_batches, y_batches):
    sess.run(
        training_op,
        feed_dict={hidden2: hidden2_batch, y: y_batch})
'''
```

```
'import numpy as np\n\nn_epochs = 100\nn_batches = 500\n\nfor ep
och in range(n_epochs):\n    shuffled_idx = rnd.permutation(\n
      len(hidden2_outputs))\n    \n    hidden2_batches = np.arra
y_split(\n        hidden2_outputs[shuffled_idx], \n        n_bat
ches)\n    \ny_batches = np.array_split(\n    y_train[shuffled_i
dx], \n    n_batches)\n\nfor hidden2_batch, y_batch in zip(hidde
n2_batches, y_batches):\n    sess.run(\n        training_op, \n
       feed_dict={hidden2: hidden2_batch, y: y_batch})\n'
```

Tweaking/Dropping/Replacing Upper Layers

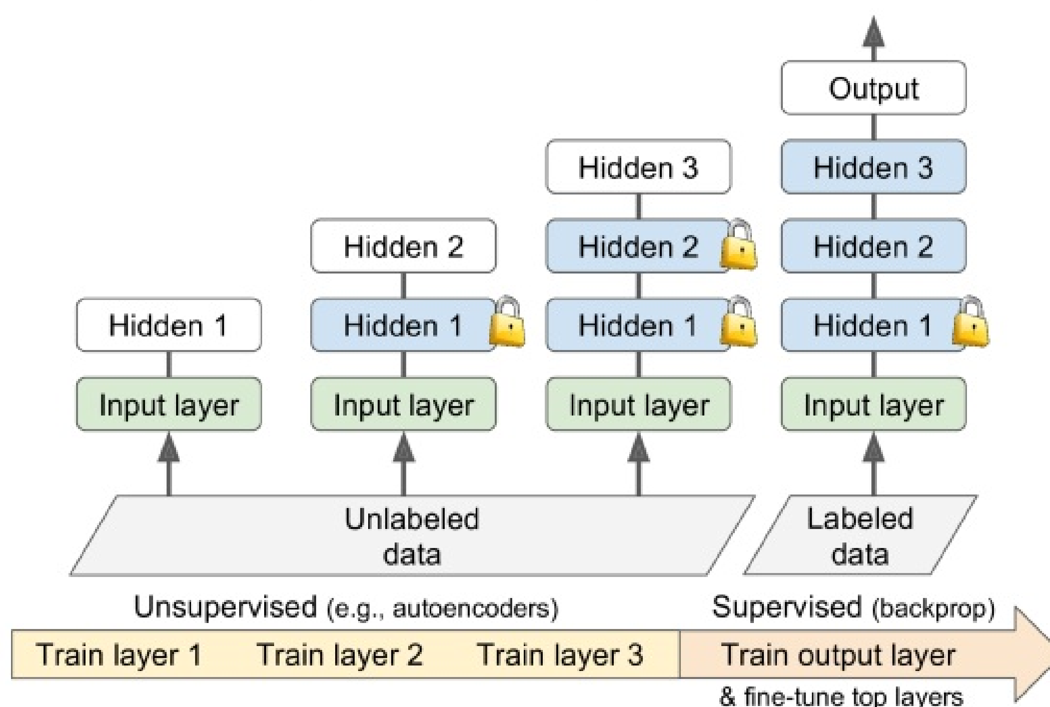
- original output layer: should be replaced (little chance of reuse)
- iterative freeze/train/compare process to see how many upper layers needed

Model Zoos

- When you want to find a net already trained on a similar task
- TensorFlow Model Zoo
- Caffe Model Zoo - converter on github

Unsupervised pre-training

- Tough problem, but doable.
- Train layers one-by-one, starting with lowest layer
- Freeze completed layers & train next layer on previous results



Pre-training on easily labeled data - reuse lower layers for "real" task

- Often required due to cost/availability of large labeled datasets
- Common tactic: label all training data as "good", generate & corrupt additional instances, label new ones as "bad.

Faster Optimizers

- Training speedup strategies thus far: 1) smart weight initializations, 2) smart

activation functions, 3) batch normalization, 4) reuse of pretraining.
- Better optimizer choices:
    - Momentum optimization
    - Nesterov Accelerated Gradients
    - AdaGrad
    - RMSProp
    - Adam (should almost always use this one)
- Worth noting: below techniques rely on 1st-order partial derivatives (Jacobians); more techniques in literature use 2nd-order derivs (Hessians). Not viable for most deep learning due to memory & computational requirements.

Momentum optimization

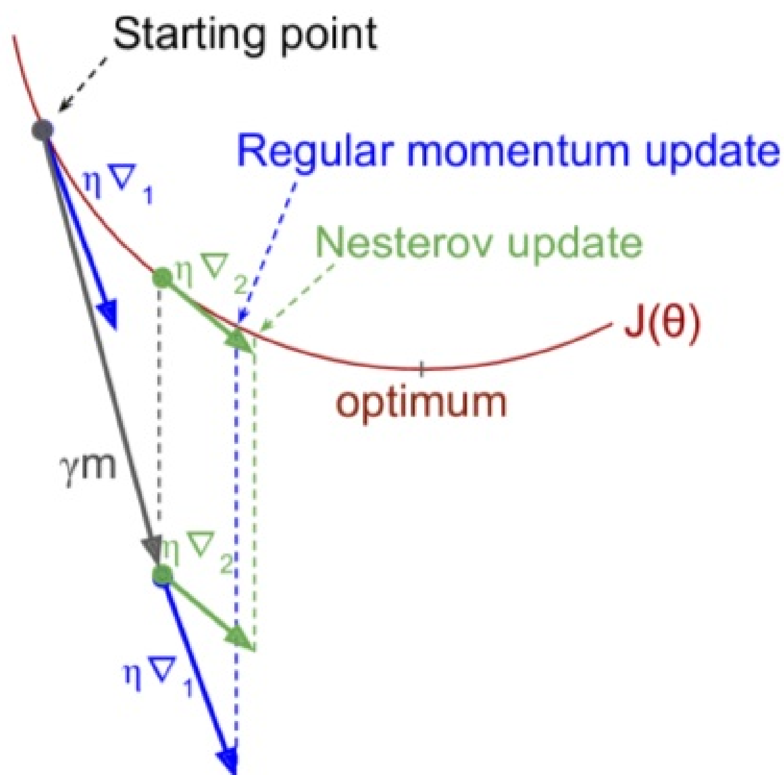
- local gradient added to a **momentum vector** (m) multiplied by learning rate (n)
- ie, **gradient used as an accelerant - not as a speed.**
- *beta* hyperparameter serves as friction mechanism. 0 = high friction, 1 = no friction.
- Momentum optimization escapes plateaus much faster than GD.

```
# in TF

optimizer = tf.train.MomentumOptimizer(
    learning_rate=learning_rate,
    momentum=0.9)
```

Nesterov Accelerated Gradient

- idea: measure cost function gradient *slightly ahead in direction of momentum*.

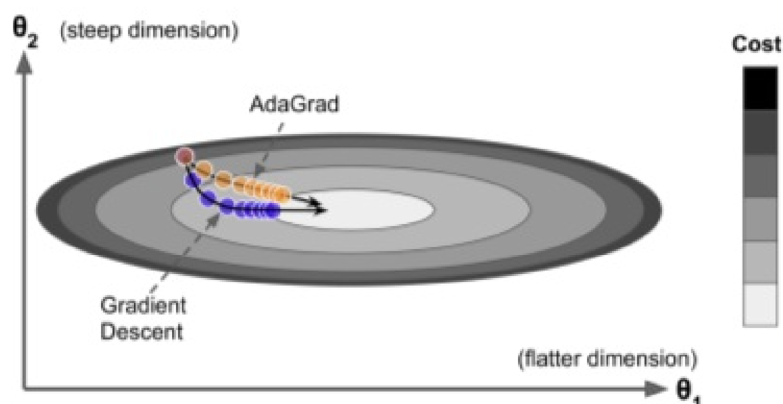



```
# in TF

optimizer = tf.train.MomentumOptimizer(
    learning_rate=learning_rate,
    momentum=0.9,
    use_nesterov=True)
```

AdaGrad

- Scales gradient vector along steepest dimensions, ie it decays the learning rate faster for steep dimensions. (ie *adaptive learning rate*)
- Works on simple quadratic problems but often stops too early.



RMSProp

- Fixes AdaGrad problem by accumulating most recent gradients (instead of all).
- Better than AdaGrad on all but very simple problems. Also better than MO and Nesterov.

```
# in TF

optimizer = tf.train.RMSPropOptimizer(
    learning_rate=learning_rate,
    momentum=0.9,
    decay=0.9,
    epsilon=1e-10)
```

Adam Optimization (paper:)

- Keeps track of decaying past gradients (like Momentum Optimization)
- Keeps track of decaying past squared gradients (like RMSProp)

- Default params in TF:

- Momentum decay param (beta1) usually set to 0.9
- Scaling decay param (beta2) usually set to 0.999
- Smoothing term (epsilon) usually set to 10e-8

```
# in TF

optimizer = tf.train.AdamOptimizer(
    learning_rate=learning_rate)
```

Learning Rate Scheduling

```
# in TF

initial_learning_rate = 0.1
decay_steps = 10000
decay_rate = 1/10

global_step = tf.Variable(
    0, trainable=False)

learning_rate = tf.train.exponential_decay(
    initial_learning_rate,
    global_step,
    decay_steps,
    decay_rate)

optimizer = tf.train.MomentumOptimizer(
    learning_rate,
    momentum=0.9)

training_op = optimizer.minimize(
    loss,
    global_step=global_step)
```

Regularization Techniques

Early Stopping

- Simply interrupt training when validation performance starts dropping.

L1 & L2 Regularlization

Dropout

- Popular technique - typically adds 1-2% accuracy boost
- At every training step, every neuron has probability (p) of being temporarily ignored
- Typical p = 50%

- In TF: apply *dropout()* to input layer & output of every hidden layer.

```
# in TF

from tensorflow.contrib.layers import dropout

[...]

is_training = tf.placeholder(
    tf.bool,
    shape=(),
    name='is_training')

keep_prob = 0.5

X_drop = dropout(
    X,
    keep_prob,
    is_training=is_training)

hidden1      = fully_connected(
    X_drop, n_hidden1, scope="hidden1")

hidden1_drop = dropout(
    hidden1, keep_prob, is_training=is_training)

hidden2      = fully_connected(
    hidden1_drop, n_hidden2, scope="hidden2")

hidden2_drop = dropout(
    hidden2, keep_prob, is_training=is_training)

logits       = fully_connected(
    hidden2_drop, n_outputs,
    activation_fn=None,
    scope="outputs")
```

Max-Norm Regularization

- Each neuron's incoming weights are constrained such that ||w||2 <= r
- r = *max-norm hyperparameter*
- ||.|| = l2 norm
- Reducing r increases regularization
- Not implemented in TF, but doable.

Data Augmentation

- Generating new training instances from existing ones with learnable differences
- ex: pics with shifts/rotates/resizes/flips/contrasts
- TF has image manipulation ops built-in

Practical Guidelines

- Suggested default DNN configurations:
  - Initialization: He
  - Activation: ELU
  - Normalization: Batch
  - Regularization: Dropout
  - Optimizer: Adam
  - Learning Rate schedule: none

Intro

Multi Devices, Single Machine

- Check if GPU cards have nVidia Compute Capability >3.0
- Alternative using AWS: helpful blog post
- Google Cloud service: uses TPU hardware
- Which to use? (Tim Dettmers)
- Download CUDA & CuDNN, set their environment vars
- use *nvidie-smi* cmnd to check installation
- install TF with GPU support
- open Python shell, verify TF detects CUDA & cuDNN

  > import tensorflow as tf
  >
  > sess = tf.Session()

```
import tensorflow as tf

config = tf.ConfigProto()
#config.gpu_options.per_process_gpu_memory_fraction=0.4
config
```

Managing GPU RAM

- TF grabs all GPU RAM on first graph invocation. To run 2nd TF program while the 1st is still running, run each process on different GPU cards. (Below: program #1 sees GPUs 0,1; program #2 sees GPUs 2,3.)

> $ CUDA_VISIBLE_DEVICES=0,1 python3 program_1.py
>
> $ CUDA_VISIBLE_DEVICES=3,2 python3 program_2.py

- Option 2: tell TF to grab a % of memory. (Below: 40% allocation.)

```
session = tf.Session(config=config)
config,session
```

Placing Ops on Devices

Parallel Execution

- TF Whitepaper - dynamic algorithm, distributes ops across all available devices. **But not available (yet) in open-source TF.**

Simple Placement

- Mostly up to you. To pin devices to specific device, use a device() function. Below: a,b pinned to cpu#0; c can go anywhere.

```
import tensorflow as tf

with tf.device("/cpu:0"):
    a,b = tf.Variable(3.0), tf.Variable(4.0)
c = a*b
c
```

Logging Placements

- Use *log_device_placement=True*. This tells placer to log msg whenever a node is "placed".

```
import tensorflow as tf

config = tf.ConfigProto()
config.log_device_placement = True
sess = tf.Session(config=config)
print(config,"\n",sess)
```

Dynamic Placement

- You can specify a function instead of a device when creating a device block.

```
import tensorflow as tf

def variables_on_cpu(op):
    if op.type == "Variable":
        return "/cpu:0"
    else:
        return "/cpu:0"
with tf.device(variables_on_cpu):
    a = tf.Variable(3.0)
    b = tf.constant(4.0)
    c = a * b
c
```

Ops & Kernels

- TF operations need to define a **kernel** to run n a device. **Not all ops have kernels for both GPUs and CPUs**. Example: TF doesn't have integer kernel for GPUs. Changin i (below) from 3 to 3.0 should allow op to run.

```
import tensorflow as tf
with tf.device("/gpu:0"):
    i = tf.Variable(3)
test = sess.run(i.initializer)
test
```

- To allow TF to "fall back" to a CPU instead, use *allow_soft_placement=True*.

```
with tf.device("/gpu:0"):
    i = tf.Variable(3)
config = tf.ConfigProto()
config.allow_soft_placement = True
sess = tf.Session(config=config)
test = sess.run(i.initializer) # the placer runs and falls back
to /cpu:0
print(test)
```

Parallel Execution

- TF executes any nodes with zero dependencies first. If those nodes are on **separate** devices, they are run in parallel. If on the same device, they are run in different threads & **may** be run in parallel.

Control Dependencies

- Use *control dependencies* to control/postpone node evaluations (ex: premature memory hogging).

```
import tensorflow as tf
a = tf.constant(1.0)
b = a + 2.0

with tf.control_dependencies([a,b]):
    x = tf.constant(3.0)
    y = tf.constant(4.0)

print(x+y)
```

Multiple Devices - Multiple Servers

- cluster: >=1 TF servers ("tasks") across machines. Tasks belong to **jobs** (collections of related tasks)
- "ps" = parameter server
- "worker" = computing engine

```
cluster_spec = tf.train.ClusterSpec({
"ps": [
"machine-a.example.com:2221", # /job:ps/task:0
],
"worker": [
"machine-a.example.com:2222", # /job:worker/task:0
"machine-b.example.com:2222", # /job:worker/task:1
]})

cluster_spec
```

```
server.join()
# blocks main thread until server stops (i.e., never)
```

Opening a Session

```
# NOT YET WORKING
# open session
#a = tf.constant(1.0)
#b = a + 2
#c = a * 3
#with tf.Session("grpc://machine-b.example.com:2222") as sess:
#    print(c.eval()) # 9.0
```

Master & Worker Services

- gRPC protocol to talk to servers. HTTP2 basis, bidirectional
- based on *protocol buffers*
- all servers can provide *master* & *worker* services.

Pinning Ops Across Tasks

- you can pin ops to any device
- ex:

```
# NOT WORKING YET
#with tf.device("/job:ps/task:0/cpu:0")
#a = tf.constant(1.0)
#with tf.device("/job:worker/task:0/cpu:0")
#with tf.device("/job:worker/task:0/gpu:1")
#b = a + 2
#c = a + b
```

Sharding Variables across Multiple Param Servers

- sharding across servers mitigates risk of network card saturation
- TF distribs variables across all "ps" tasks - round robin setup

```
'''NOT WORKING YET
import tensorflow as tf
with tf.device(tf.train.replica_device_setter(ps_tasks=2):
    v1 = tf.Variable(1.0) # pinned to /job:ps/task:0
    v2 = tf.Variable(2.0) # pinned to /job:ps/task:1
    v3 = tf.Variable(3.0) # pinned to /job:ps/task:0
    v4 = tf.Variable(4.0) # pinned to /job:ps/task:1
    v5 = tf.Variable(5.0) # pinned to /job:ps/task:0
'''
```

```
'NOT WORKING YET\nimport tensorflow as tf\nwith tf.device(tf.tra
in.replica_device_setter(ps_tasks=2):\n    v1 = tf.Variable(1.0)
 # pinned to /job:ps/task:0\n    v2 = tf.Variable(2.0) # pinned
to /job:ps/task:1\n    v3 = tf.Variable(3.0) # pinned to /job:ps
/task:0\n    v4 = tf.Variable(4.0) # pinned to /job:ps/task:1\n
   v5 = tf.Variable(5.0) # pinned to /job:ps/task:0\n'
```

Sharing State across Sessions (Resource Containers)

- local session: all vars managed by session itself & vanish on end.
- distributed session: vars managed by *resource containers* on cluster

```
'''# simple_client.py
#import tensorflow as tf
#import sys
#x = tf.Variable(0.0, name="x")
#increment_x = tf.assign(x, x + 1)
#with tf.Session(sys.argv[1]) as sess:
#    if sys.argv[2:]==["init"]:
#sess.run(x.initializer)
#sess.run(increment_x)
#print(x.eval())
'''
```

```
'# simple_client.py\n#import tensorflow as tf\n#import sys\n#x =
 tf.Variable(0.0, name="x")\n#increment_x = tf.assign(x, x + 1)\
n#with tf.Session(sys.argv[1]) as sess:\n#    if sys.argv[2:]==[
"init"]:\n#sess.run(x.initializer)\n#sess.run(increment_x)\n#pri
nt(x.eval())\n'
```

```
# launches client which connects to B, reuses variable x
# python3 simple_client.py grpc://machine-b.example.com:2222
#2.0
```

Async Communications (TF Queues)

- Queueing data
- DeQueueing data
- Queues of tuples
- Closing a queue
- RandomShuffleQueue
- PaddingFifoQueue

Loading Data Directly from Graph

- Needed to avoid file server (bandwidth) saturation
- Preloading data to variables

- Reading data from graph with **reader operations**
    - CSV, binary, TFRecords
    - **TextLineReader** reads file lines one-by-one
    - record identifier (string): filename:linenumber
    - tf.decode_csv(val, record_defaults=[...])

```
'''TO LOAD A GRAPH
instance_queue = tf.RandomShuffleQueue(
    capacity=10,
    min_after_dequeue=2,
    dtypes=[tf.float32, tf.int32],
    shapes=[[2],[]],
    name="instance_q",
    shared_name="shared_instance_q")

enqueue_instance = instance_queue.enqueue([features, target])
close_instance_queue = instance_queue.close()
'''
```

```
'TO LOAD A GRAPH\ninstance_queue = tf.RandomShuffleQueue(\n    c
apacity=10, \n    min_after_dequeue=2,\n    dtypes=[tf.float32,
tf.int32], \n    shapes=[[2],[]],\n    name="instance_q", \n
shared_name="shared_instance_q")\n\nenqueue_instance = instance_
queue.enqueue([features, target])\nclose_instance_queue = instan
ce_queue.close()\n'
```

```
'''TO RUN THE GRAPH
with tf.Session([...]) as sess:
    sess.run(enqueue_filename, feed_dict={filename: "my_test.csv
"})
    sess.run(close_filename_queue)
    try:
        while True:
            sess.run(enqueue_instance)
    except tf.errors.OutOfRangeError as ex:
        pass # no more records in the current file and no more f
iles to read
    sess.run(close_instance_queue)
'''
```

```
'TO RUN THE GRAPH\nwith tf.Session([...]) as sess:\n    sess.run
(enqueue_filename, feed_dict={filename: "my_test.csv"})\n    ses
s.run(close_filename_queue)\n    try:\n        while True:\n
        sess.run(enqueue_instance)\n    except tf.errors.OutOfRa
ngeError as ex:\n        pass # no more records in the current f
ile and no more files to read\n    sess.run(close_instance_queue
)\n'
```

Multithreaded readers using a Coordinator & QueueRunner

Other convenience functions

- *string_input_producer()*
- *tf.train.start_queue_runners()*

  producer functions = create queues

- input_producer()
- range_input_producer()
- slice_input_procucer()
- shuffle_batch(list_of_tensors)
  - returns RandomShuffleQueue
  - returns QueueRunner (added to GraphKeys.QUEUE_RUNNERS)

  - dequeue_many() = returns minibatch from queue
  - batch() --?
  - batch_join() --?
  - shuffle_batch_join() --?

One NN per Device

- near-linear speedup: training 100 nets across 50 servers x 2 gpus/server roughly equiv to 1 net on 1 gpu. (**perfect for hyperparamer tuning**)
- potential option: tf serving, released 2/2016

In-Graph vs Between-Graph Replication (for Ensembles)

- Two approaches to building ensembles:

1) one big graph, one session, any server in cluster ("in graph replication")

2) one graph/network, handle synchronization yourself ("between graph replication") using queues -- considered more flexible

```
#RunOptions ... timeout_in_ms()
'''NOT YET
with tf.Session([...]) as sess:
    [...]
    run_options = tf.RunOptions()
    run_options.timeout_in_ms = 1000 # 1s timeout
    try:
        pred = sess.run(dequeue_prediction, options=run_options)
    except tf.errors.DeadlineExceededError as ex:
        [...] # the dequeue operation timed out after 1s
'''
```

```
'NOT YET\nwith tf.Session([...]) as sess:\n    [...]\n    run_options = tf.RunOptions()\n    run_options.timeout_in_ms = 1000 # 1s timeout\n    try:\n        pred = sess.run(dequeue_prediction, options=run_options)\n    except tf.errors.DeadlineExceededError as ex:\n        [...] # the dequeue operation timed out after 1s\n'
```

```
#
'''NOT YET
config = tf.ConfigProto()
config.operation_timeout_in_ms = 1000
# 1s timeout for every operation
with tf.Session([...], config=config) as sess:
    [...]
    try:
        pred = sess.run(dequeue_prediction)
    except tf.errors.DeadlineExceededError as ex:
        [...] # the dequeue operation timed out after 1s
'''
```

```
'NOT YET\nconfig = tf.ConfigProto()\nconfig.operation_timeout_in_ms = 1000\n# 1s timeout for every operation\nwith tf.Session([...], config=config) as sess:\n    [...]\n    try:\n        pred = sess.run(dequeue_prediction)\n    except tf.errors.DeadlineExceededError as ex:\n        [...] # the dequeue operation timed out after 1s\n'
```

Model Parallelism

- Chopping models, running chunks on different devices
- Fully Connect Nets (FCNs): not much value in doing this
- Vertical & Horiz slicing don't work well either

- Nets w/ partially connected layers (CNNs): easier to distribute

- Some RNNs use mem cells (input from own output at t+1)

Data Parallelism

- **Sync updates** (aggregator waits for all gradients to be available, finds avg, applies result) - could be delayed by slow devices; params could also saturate server bandwidth
- **Async updates** - more training steps/minute. issue: "stale gradients" (when computing gradients falls behind rate of parameter change) - slows convergence, introduces noise/wobble. To avoid this:
  - reduce learning rate
  - drop/scaleback stale gradients
  - adjust minibatch size
  - Start first few epochs with just one replica ("warmup phase")
- **Bandwidth** - At some point, more GPUs doesn't help because network saturation won't allow more data traffic. google report. Steps you can take:
  - group gpus on single server (avoids network hops)
  - shard params acrosss servers
  - drop precision from float32 to bfloat16
  - 8b precision ("quantization"): see mobile phone apps
- **How TF does it** -
  - you choose 1) replication type (in-graph, between-graph) and 2) update type (async or sync) 1) in-graph + sync: one big graph 2) in-graph + async: 1 optimizer/replica, 1 thread/replica 3) bw-graph + sync: wrap optimizer in **SyncReplicasOptimizer**

Intro - Visual Cortex

- LeNet-5 paper - intro'd convo & pooling layers

```
%%html
<style>
table,td,tr,th {border:none!important}
</style>
```

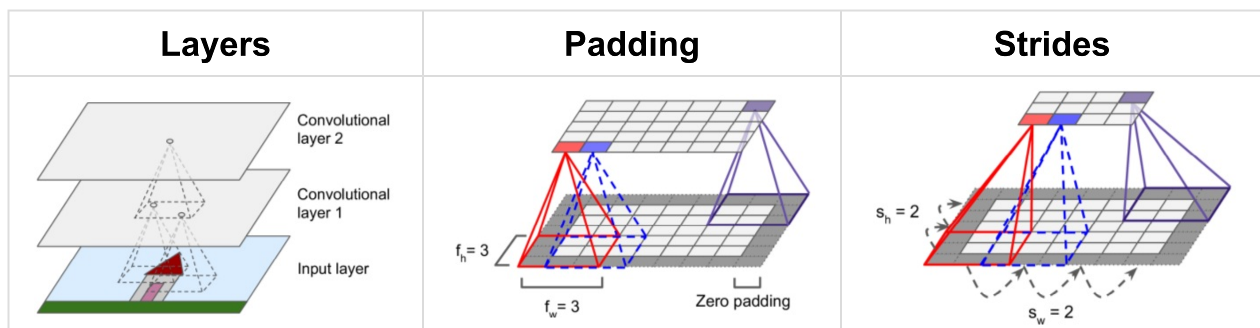
```
# utilities
import matplotlib.pyplot as plt

def plot_image(image):
    plt.imshow(image, cmap="gray", interpolation="nearest")
    plt.axis("off")

def plot_color_image(image):
    plt.imshow(image.astype(np.uint8),interpolation="nearest")
    plt.axis("off")
```

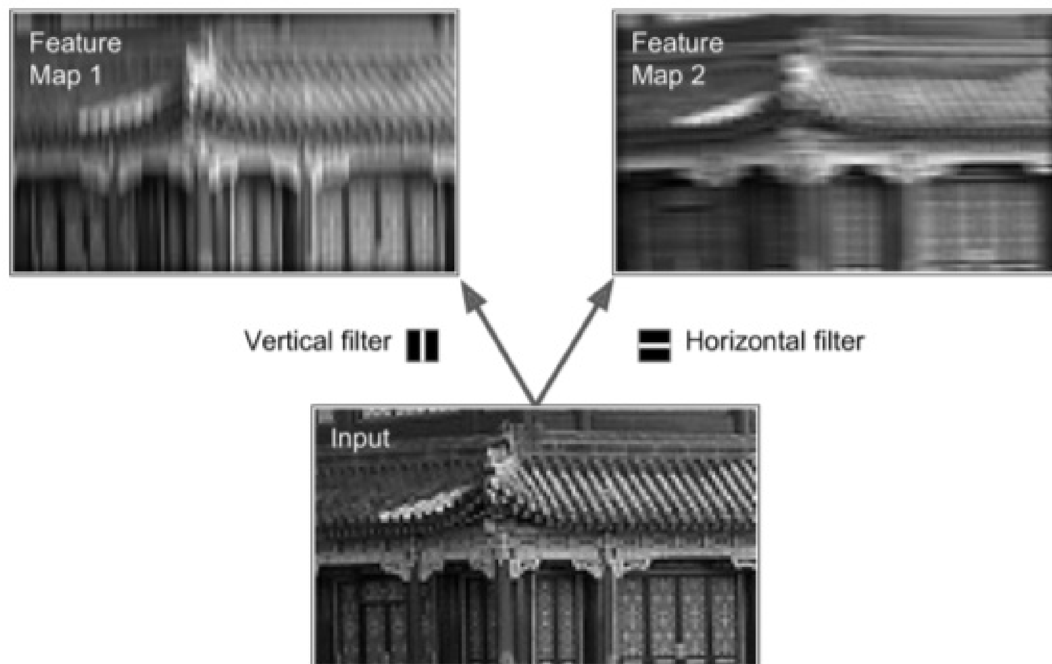
Convolutional Layers

- math detail
- neurons connected to receptor field in next layer. uses *zero padding* to force layers to have same height & width.
- also can connect large input layer to much smaller layer by spacing out receptor fields (distance between receptor fields = *stride*)



Filters

- neuron weights can look like small image (w/ size = receptor field)
- examples given: 1) vertical filter (single vertical bar, mid-image, all other cells zero) 2) horizontal filter (single horizontal bar, mid-image, all other cells zero)
- both return **feature maps** (highlights areas of image most similar to filter)



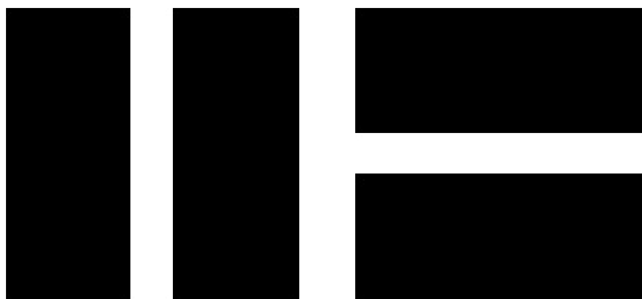

```
import numpy as np

fmap = np.zeros(shape=(7, 7, 1, 2), dtype=np.float32)
fmap[:, 3, 0, 0] = 1
fmap[3, :, 0, 1] = 1
print(fmap[:, :, 0, 0])
print(fmap[:, :, 0, 1])

plt.figure(figsize=(6,6))

plt.subplot(121)
plot_image(fmap[:, :, 0, 0])
plt.subplot(122)
plot_image(fmap[:, :, 0, 1])
plt.show()
```

```
[[ 0.  0.  0.  1.  0.  0.  0.]
 [ 0.  0.  0.  1.  0.  0.  0.]
 [ 0.  0.  0.  1.  0.  0.  0.]
 [ 0.  0.  0.  1.  0.  0.  0.]
 [ 0.  0.  0.  1.  0.  0.  0.]
 [ 0.  0.  0.  1.  0.  0.  0.]
 [ 0.  0.  0.  1.  0.  0.  0.]]
[[ 0.  0.  0.  0.  0.  0.  0.]
 [ 0.  0.  0.  0.  0.  0.  0.]
 [ 0.  0.  0.  0.  0.  0.  0.]
 [ 1.  1.  1.  1.  1.  1.  1.]
 [ 0.  0.  0.  0.  0.  0.  0.]
 [ 0.  0.  0.  0.  0.  0.  0.]
 [ 0.  0.  0.  0.  0.  0.  0.]]
```



```
from sklearn.datasets import load_sample_image

china = load_sample_image("china.jpg")
flower = load_sample_image("flower.jpg")

image = china[150:220, 130:250]
height, width, channels = image.shape

image_grayscale = image.mean(axis=2).astype(np.float32)
images = image_grayscale.reshape(1, height, width, 1)
```

```
import tensorflow as tf

tf.reset_default_graph()

# Define the model

X = tf.placeholder(
    tf.float32,
    shape=(None, height, width, 1))

feature_maps = tf.constant(fmap)

convolution = tf.nn.conv2d(
    X,
    feature_maps,
    strides=[1,1,1,1],
    padding="SAME",
    use_cudnn_on_gpu=False)
```

```
# Run the model

with tf.Session() as sess:
    output = convolution.eval(feed_dict={X: images})
```

```
plt.figure(figsize=(6,6))

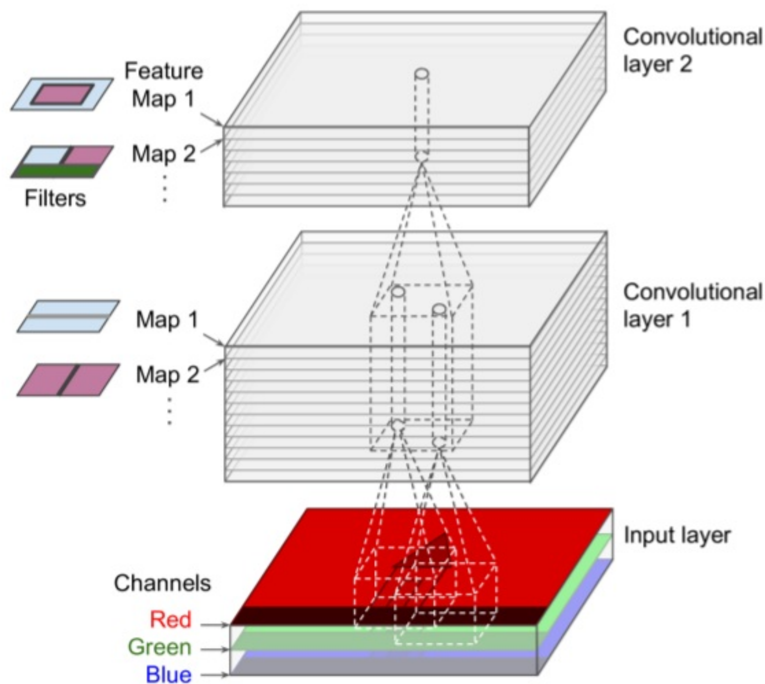
#plt.subplot(121)
plot_image(images[0, :, :, 0])
#plt.subplot(122)
plot_image(output[0, :, :, 0])
#plt.subplot(123)
plot_image(output[0, :, :, 1])
plt.show()
```



```
%%html
<style>
img[alt=stacking] { width: 400px; }
</style>
```

Stacking Feature Maps

- images made of *sublayers* (one per color channel, typical red/green/blue, grayscale = one chan, others = many chans)



```
import numpy as np
from sklearn.datasets import load_sample_images

# Load sample images
dataset = np.array(load_sample_images().images, dtype=np.float32
)
batch_size, height, width, channels = dataset.shape

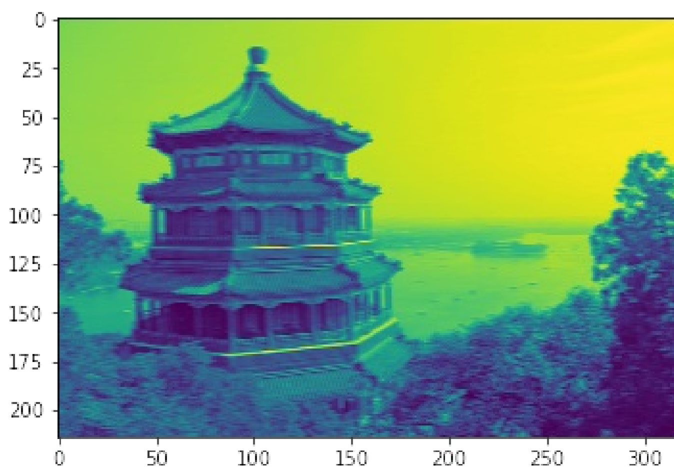
# Create 2 filters
filters = np.zeros(shape=(7, 7, channels, 2), dtype=np.float32)
filters[:, 3, :, 0] = 1 # vertical line
filters[3, :, :, 1] = 1 # horizontal line

# Create a graph with input X plus a convolutional layer applyin
g the 2 filters
X = tf.placeholder(tf.float32,
                   shape=(None, height, width, channels))

convolution = tf.nn.conv2d(
    X, filters, strides=[1,2,2,1], padding="SAME")

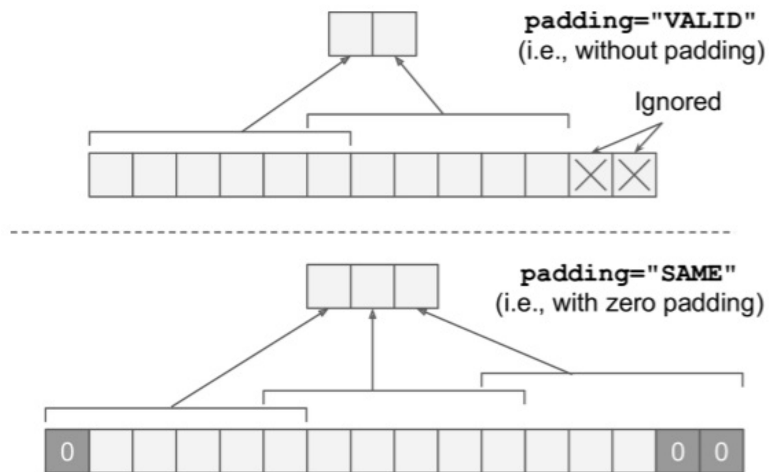
with tf.Session() as sess:
    output = sess.run(convolution, feed_dict={X: dataset})

plt.imshow(output[0, :, :, 1])
plt.show()
```



```
%%html
<style>
img[alt=padding] { width: 400px; }
</style>
```

"Valid" v. "Same" Padding



```
import tensorflow as tf
import numpy as np

tf.reset_default_graph()

filter_primes = np.array(
    [2., 3., 5., 7., 11., 13.],
    dtype=np.float32)

x = tf.constant(
    np.arange(1, 13+1, dtype=np.float32).reshape([1, 1, 13, 1]))

print ("x:\n",x)

filters = tf.constant(
    filter_primes.reshape(1, 6, 1, 1))

# conv2d arguments:
# x = input minibatch = 4D tensor
# filters = 4D tensor
# strides = 1D array (1, vstride, hstride, 1)
# padding = VALID = no zero padding, may ignore edge rows/cols
# padding = SAME  = zero padding used if needed

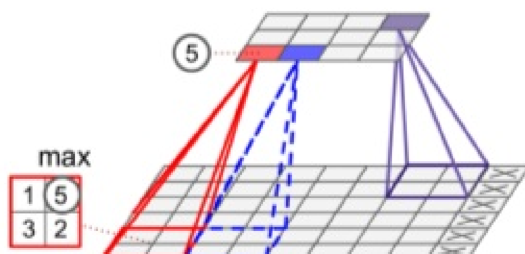
valid_conv = tf.nn.conv2d(x, filters, strides=[1, 1, 5, 1], padd
ing='VALID')
same_conv =  tf.nn.conv2d(x, filters, strides=[1, 1, 5, 1], padd
ing='SAME')

with tf.Session() as sess:
    print("VALID:\n", valid_conv.eval())
    print("SAME:\n", same_conv.eval())
```

```
x:
 Tensor("Const:0", shape=(1, 1, 13, 1), dtype=float32)
VALID:
 [[[[ 184.]
   [ 389.]]]]
SAME:
 [[[[ 143.]
   [ 348.]
   [ 204.]]]]
```

Pooling Layers

- Goal: subsample (shrink) input image to reduce loading.
- Need to define pool size, stride & padding type.
- Result: aggregation function (max, mean)
- Below: max pool, 2x2, stride = 2, no padding.



```
dataset = np.array([china, flower], dtype=np.float32)

batch_size, height, width, channels = dataset.shape

filters = np.zeros(shape=(7, 7, channels, 2), dtype=np.float32)
filters[:, 3, :, 0] = 1  # vertical line
filters[3, :, :, 1] = 1  # horizontal line

X = tf.placeholder(tf.float32,
                   shape=(None, height, width, channels))

# alternative: avg_pool()

max_pool = tf.nn.max_pool(
    X,
    ksize=[1, 2, 2, 1],
    strides=[1,2,2,1],
    padding="VALID")

with tf.Session() as sess:
    output = sess.run(max_pool, feed_dict={X: dataset})

plt.figure(figsize=(12,12))
plt.subplot(121)
plot_color_image(dataset[0])
plt.subplot(122)
plot_color_image(output[0])
plt.show()
```




Memory Requirements

- Main memory killer: reverse pass of backprop - needs all intermediate vals computed during forward pass
- Example CNN:
    - 5x5 filters outputting 200 feature maps (size 150,100)
    - stride = 1, "SAME" padding
    - If image = 150x100x3 (RGB), then
    - params_count = (5x5x3+1) * 200 = 15,200
    - 200 feature maps contain 150 x 100 neurons => each needs to compute weighted sum of 5x5x3 = 75 inputs => 225M floating-point multiplies.
    - If using 32b float => output requires 200x150x100x32 = 96M bits = 11.4MB for one instance

During inference: one layer's memory can be dropped when next layer is computed. (You only need enough memory for two layers).

During training: all computed values have to preserved for reverse pass (You need enough memory for all layers.)

CNN Architectures

LeNet-5 (c. 1998, used to solve MNIST digits dataset)

Layer	Type	Maps	Size	Kernel size	Stride	Activation
Out	Fully Connected	–	10	–	–	RBF
F6	Fully Connected	–	84	–	–	tanh
C5	Convolution	120	1 × 1	5 × 5	1	tanh
S4	Avg Pooling	16	5 × 5	2 × 2	2	tanh
C3	Convolution	16	10 × 10	5 × 5	1	tanh
S2	Avg Pooling	6	14 × 14	2 × 2	2	tanh
C1	Convolution	6	28 × 28	5 × 5	1	tanh
In	Input	1	32 × 32	–	–	–

- MNIST images zero-padded to 32x32 & normalized
- pooling layers: mean x learned coefficient + learnable bias
- output layer: output = Euclidian distance (input vect, weight vect)

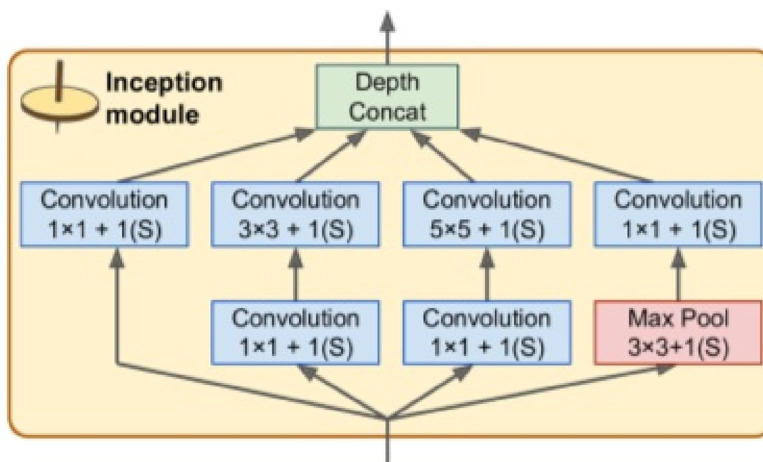
AlexNet won ILSVRC 2012

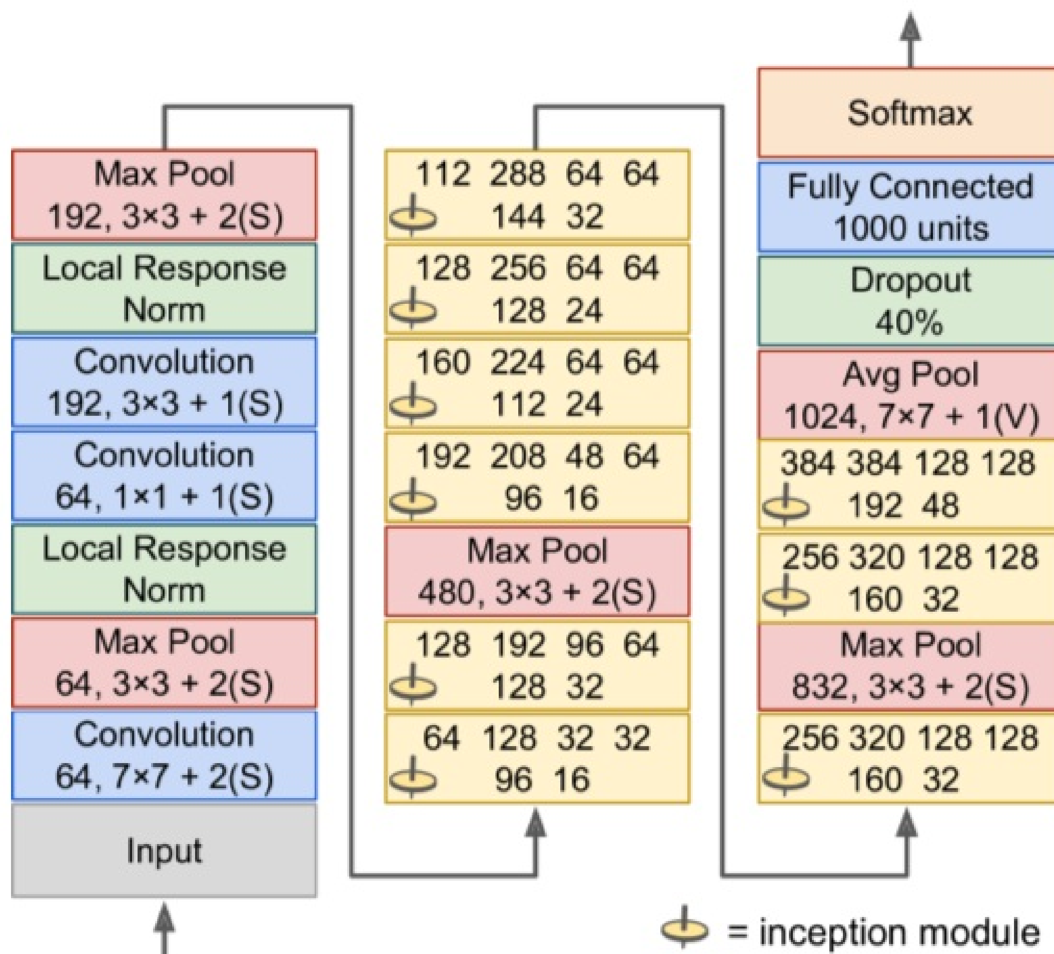
Layer	Type	Maps	Size	Kernel size	Stride	Padding	Activation
Out	Fully Connected	–	1,000	–	–	–	Softmax
F9	Fully Connected	–	4,096	–	–	–	ReLU
F8	Fully Connected	–	4,096	–	–	–	ReLU
C7	Convolution	256	13 × 13	3 × 3	1	SAME	ReLU
C6	Convolution	384	13 × 13	3 × 3	1	SAME	ReLU
C5	Convolution	384	13 × 13	3 × 3	1	SAME	ReLU
S4	Max Pooling	256	13 × 13	3 × 3	2	VALID	–
C3	Convolution	256	27 × 27	5 × 5	1	SAME	ReLU
S2	Max Pooling	96	27 × 27	3 × 3	2	VALID	–
C1	Convolution	96	55 × 55	11 × 11	4	SAME	ReLU
In	Input	3 (RGB)	224 × 224	–	–	–	–

- Uses 50% dropout on layers F8, F9 for regularization
- Uses random image shifts/flips/rotates/lighting to augment dataset
- Uses *local response normalization* on layers C1, C3.
- Hyperparameter settings: r=2, alpha=0.00002, beta=0.75, k=1
- ZFNet (tweaked AlexNet) won ILSVRC 2013.

GoogLeNet won ILSVRC 2014

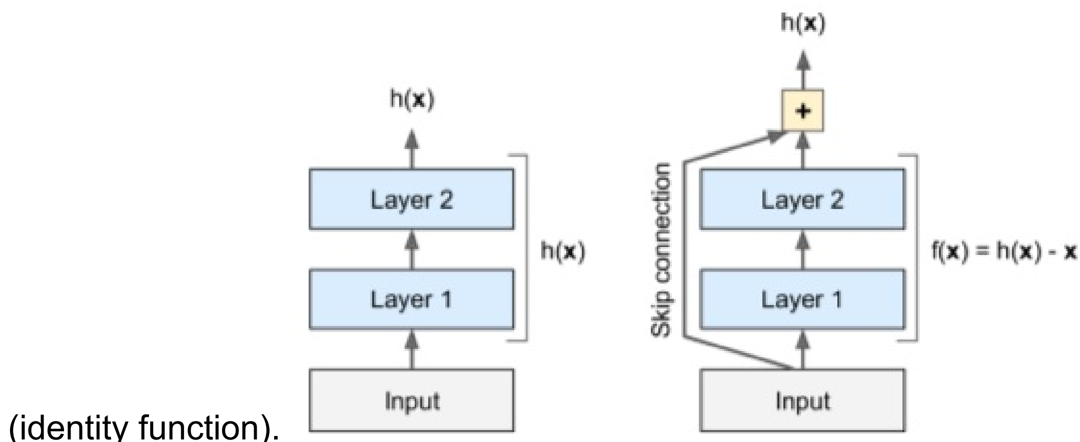
- Much deeper than previous nets
- Uses *inception modules* to use params much more efficiently. They use 1x1 kernels as "bottleneck layers" (reduces dimensionality). Also: pairs of convo layers act as single more powerful convo layer.
- All convo layers use ReLU activation.



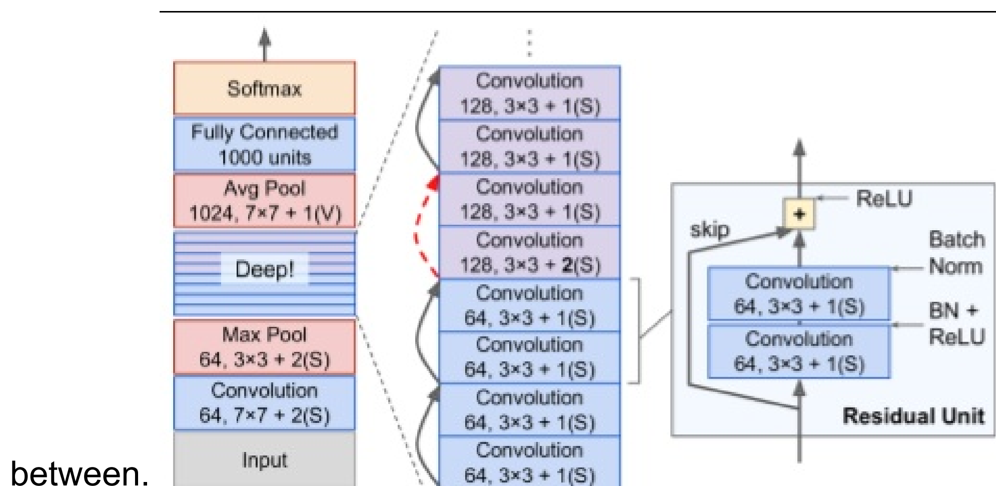


ResNet

- 152 layers deep
- Uses *skip connections* to connect non-adjacent layers in stack
- skip connections force learning model f(x) = h(x) - x (*residual learning*). When initialized, weights near zero => network outputs values near-copy of inputs



(identity function).
- architecture: stack starts & ends like GoogLeNet, stack of residual units in

between. 

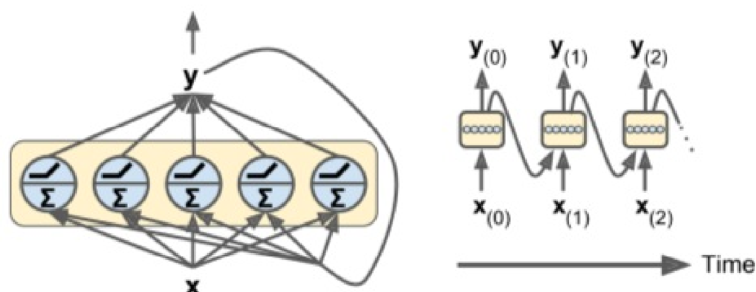
TF Convolution Ops

- conv1d() - 1D layer - good for NLP
- conv3d() - 3D layer - good for PET scans
- atrous_conv2D() - 2D layer with "holes"
- conv2d_transpose() - 2D "deconvolutional layer" - upsamples image by inserting zeroes * between inputs
- depthwise_conv2d() - applies every filter to each input channel independently
- separable_conv2d() - depthwise convo, then apply 1x1 CNN layer to result

```
%%html
<style>
img[alt=recurrent_unrolled] { width: 400px; }
</style>
<style>
img[alt=sequence_vector] { width: 400px; }
</style>
<style>
img[alt=gru-cell] { width: 400px; }
</style>
<style>
img[alt=encoder-decoder] { width: 400px; }
</style>
```

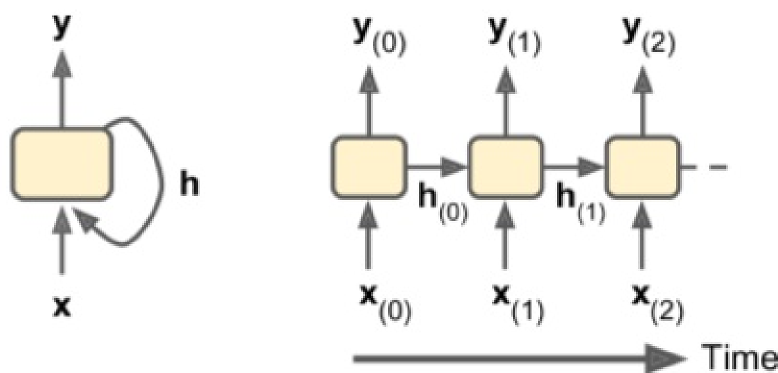
Intro

- Use case: arbitrary-length **sequence** data analysis - *anticipation* abilities
- RNNs much like feed-forward NNs, but also with backward-facing connections
- At time step *t* each node sees input *x(t)* plus its previous output *y(t-1)*.
- Below: "unrolling" a net across a time axis.



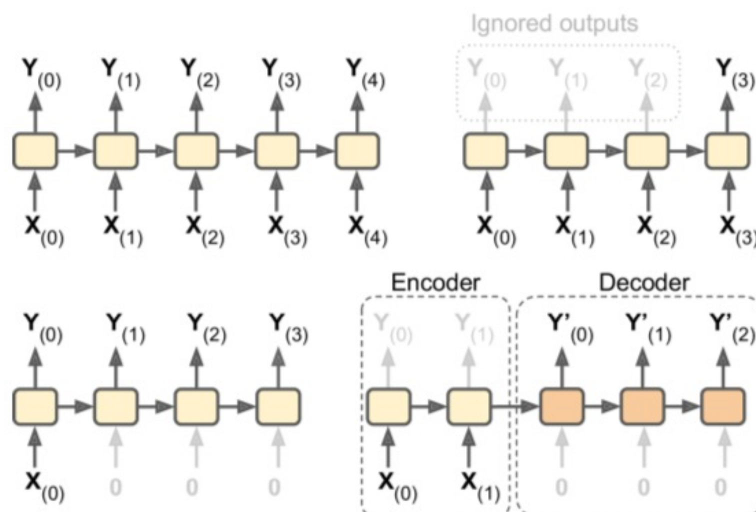
Memory Cells

- A network node that preserves state across time is called a **cell** (memory cell).
- *h(t)* is a cell's "hidden" state at time=t.



Input/Output Sequences

- RNNs can be used to predict the results of time shifts (sequence-to-sequence), a sentiment score (sequence-to-vector), or image caption (vector-to-sequence).
- sequence-to-vector nets = **encoders**; vector-to-sequence nets = **decoders**. One use case: language translation.
- Below:
  - Top Left: Sequence-to-sequence
  - Top Right: Sequence-to-vector
  - Bot Left: Vector-to-sequence
  - Bot Right: Delayed-sequence-to-sequence



Basic RNNs in TF

- RNN design: layer of **5 recurrent cells** with tanh activation; runs over **2** time steps, and uses **vectors of size=3** at each step.

```
import tensorflow as tf

n_inputs = 3
n_neurons = 5

# two-layer net

X0 = tf.placeholder(tf.float32, [None, n_inputs])
X1 = tf.placeholder(tf.float32, [None, n_inputs])

Wx = tf.Variable(tf.random_normal(shape=[n_inputs, n_neurons],dt
ype=tf.float32))
Wy = tf.Variable(tf.random_normal(shape=[n_neurons,n_neurons],dt
ype=tf.float32))

b = tf.Variable(tf.zeros([1, n_neurons], dtype=tf.float32))

Y0 = tf.tanh(tf.matmul(X0, Wx) + b)
Y1 = tf.tanh(tf.matmul(Y0, Wy) + tf.matmul(X1, Wx) + b)

init = tf.global_variables_initializer()
```

```
# to feed inputs at both time steps,

import numpy as np
# Mini-batch: instance 0,instance 1,instance 2,instance 3

X0_batch = np.array([[0, 1, 2], [3, 4, 5], [6, 7, 8], [9, 0, 1]]
) # t = 0
X1_batch = np.array([[9, 8, 7], [0, 0, 0], [6, 5, 4], [3, 2, 1]]
) # t = 1
```

```
# Y0, Y1 = network outputs at both time steps

with tf.Session() as sess:
    init.run()
    Y0_val, Y1_val = sess.run([Y0, Y1], feed_dict={X0: X0_batch,
 X1: X1_batch})

print("output at t=0:\n",Y0_val,"\n","output at t=1\n",Y1_val)
```

```
output at t=0:
 [[-0.77183092 -0.99924457  0.23752896 -0.63130957 -0.83723265]
 [-0.92028087 -1.          0.99004787 -0.87230623 -0.99995315]
 [-0.97358704 -1.          0.999919   -0.95966864 -1.        ]
 [ 0.99999094 -0.99890459  0.9991411   0.99996841 -0.99999803]]
 output at t=1
 [[ 0.99512661 -1.          0.99997395 -0.99830353 -1.        ]
 [ 0.99977976  0.99013239 -0.96352106 -0.99476629  0.97579277]
 [ 0.99981618 -0.99989575  0.99114233 -0.99827981 -0.99984008]
 [ 0.54805535 -0.84061396 -0.99912792 -0.47432473 -0.99921536]]
```

Unrolling through Time (Static) using static_rnn()

```
tf.reset_default_graph()

n_inputs = 3
n_neurons = 5

X0 = tf.placeholder(tf.float32, [None, n_inputs])
X1 = tf.placeholder(tf.float32, [None, n_inputs])

# BasicRNNCell() -- memcell "factory"

basic_cell = tf.contrib.rnn.BasicRNNCell(
    num_units=n_neurons)

# static_rnn() -- creates unrolled RNN net by chaining cells.
# returns 1) python list of output tensors for each time step
#         2) tensor of final network states

output_seqs, states = tf.contrib.rnn.static_rnn(
    basic_cell,
    [X0, X1],
    dtype=tf.float32)

Y0, Y1 = output_seqs

init = tf.global_variables_initializer()
```

```
# to feed inputs at both time steps,

import numpy as np
# Mini-batch: instance 0,instance 1,instance 2,instance 3

X0_batch = np.array([[0, 1, 2], [3, 4, 5], [6, 7, 8], [9, 0, 1]]
) # t = 0
X1_batch = np.array([[9, 8, 7], [0, 0, 0], [6, 5, 4], [3, 2, 1]]
) # t = 1
```

```
# Y0, Y1 = network outputs at both time steps

with tf.Session() as sess:
    init.run()
    Y0_val, Y1_val = sess.run([Y0, Y1], feed_dict={X0: X0_batch,
 X1: X1_batch})

print("output at t=0:\n",Y0_val,"\n","output at t=1\n",Y1_val)
```

```
output at t=0:
 [[ 0.42442048  0.92431569 -0.2353479  -0.90074939 -0.94408685]
 [ 0.73783255  0.98977458 -0.72123086 -0.99919385 -0.99999249]
 [ 0.89336294  0.99865782 -0.9186905  -0.99999398 -1.        ]
 [-0.99143326 -0.99993676 -0.37607926  0.88796568 -0.99899191]]
 output at t=1
 [[ 0.81709599  0.48319042 -0.96708876 -0.9998284  -1.        ]
 [-0.18962485 -0.81231028 -0.21763545  0.88739753  0.57306314]
 [ 0.17130674 -0.6411857  -0.86380148 -0.95413983 -0.99999553]
 [-0.07749119 -0.86547101 -0.00461033 -0.91877526 -0.99582738]]
```

Simplification

```
tf.reset_default_graph()

n_steps = 2
n_inputs = 3
n_neurons = 5

# this time, use placeholder with add'l dimension for #timesteps
#X0 = tf.placeholder(tf.float32, [None, n_inputs])
#X1 = tf.placeholder(tf.float32, [None, n_inputs])
X =   tf.placeholder(tf.float32, [None, n_steps, n_inputs])

#print(X)

# transpose - make time steps = 1st dimension
# unstack - extract list of tensors
```

```
X_seqs = tf.unstack(
    tf.transpose(
        X, perm=[1, 0, 2]))

#print(X_seqs)

# BasicRNNCell() -- memcell "factory"

basic_cell = tf.contrib.rnn.BasicRNNCell(
    num_units=n_neurons)

# static_rnn() -- creates unrolled RNN net by chaining cells.
# returns 1) python list of output tensors for each time step
#         2) tensor of final network states

output_seqs, states = tf.contrib.rnn.static_rnn(
    basic_cell,
    X_seqs,
    dtype=tf.float32)

#Y0, Y1 = output_seqs

# stack - merge output tensors
# transpose - swap 1st two dimensions
# returns tensor shape [none, #steps, #neurons]

outputs = tf.transpose(
    tf.stack(output_seqs),
    perm=[1,0,2])

init = tf.global_variables_initializer()
```

```
X_batch = np.array([
        # t = 0      t = 1
        [[0, 1, 2], [9, 8, 7]], # instance 1
        [[3, 4, 5], [0, 0, 0]], # instance 2
        [[6, 7, 8], [6, 5, 4]], # instance 3
        [[9, 0, 1], [3, 2, 1]], # instance 4
    ])

with tf.Session() as sess:
    init.run()
    outputs_val = outputs.eval(feed_dict={X: X_batch})

print(outputs_val)
```

```
[[[ 0.76157701  0.11581181  0.64773971 -0.79434019 -0.86054337]
  [ 0.99998951 -0.66595364  0.99812627 -1.          0.84574401]]

 [[ 0.99683905  0.29572889  0.98365188 -0.99992883 -0.88169324]
  [ 0.41841054 -0.92049074 -0.64612901 -0.73361856  0.29283327]]

 [[ 0.99996316  0.45685658  0.99936479 -1.         -0.89980829]
  [ 0.99907684 -0.87088716  0.94328976 -0.9999997   0.87934762]]

 [[ 0.12318966  0.02264917  0.99982244 -0.99998975  0.99996465]
  [ 0.9525854  -0.56515652  0.08665188 -0.99705428  0.87525886]]
]
```

- Above code still not ideal - builds graph with one cell per time step. Ugly & can cause Out Of Memory errors.

Unrolling through Time using dynamic_rnn()

- uses while_loop() to iterate over the memcell
- set swap_memory=True to move GPU memory to CPU during backprop if needed
- accepts single tensor, outputs single tensor - no stack/unstack/transpose ops required.

```
tf.reset_default_graph()

X = tf.placeholder(tf.float32, [None, n_steps, n_inputs])

basic_cell = tf.contrib.rnn.BasicRNNCell(
    num_units=n_neurons)

outputs, states = tf.nn.dynamic_rnn(
    basic_cell, X, dtype=tf.float32)

init = tf.global_variables_initializer()

with tf.Session() as sess:
    init.run()
    outputs_val = outputs.eval(feed_dict={X: X_batch})

print(outputs_val)
```

```
[[[ 0.01341763 -0.10483158 -0.94257653  0.83843452 -0.20272173]
  [ 0.99978089 -0.63150525 -0.99999148  0.99999386 -0.87993085]]

 [[ 0.94205797 -0.13386673 -0.9997741   0.99812031 -0.64444101]
  [-0.6134249  -0.55738503  0.39783546  0.89031053  0.04465704]]

 [[ 0.99817288 -0.16267382 -0.99999928  0.99997997 -0.86824256]
  [ 0.99097538 -0.61533296 -0.99695957  0.99986053 -0.64558744]]

 [[ 0.9963541   0.23641461  0.75174934  0.98267573 -0.97034496]
  [ 0.85169196 -0.07830215 -0.3604137   0.95550352  0.12307668]]
]
```

Variable-Length Input Sequences

- Most problems will have variable length inputs (like sentences).
- This option uses **sequence_length** param (1D tensor)

```
tf.reset_default_graph()

X = tf.placeholder(tf.float32, [None, n_steps, n_inputs])

seq_length = tf.placeholder(tf.int32, [None])

basic_cell = tf.contrib.rnn.BasicRNNCell(
    num_units=n_neurons)

outputs, states = tf.nn.dynamic_rnn(
    basic_cell, X, dtype=tf.float32,
    #
    #
    sequence_length=seq_length)
    #
    #
X_batch = np.array([
        [[0, 1, 2], [9, 8, 7]], # instance 1
        [[3, 4, 5], [0, 0, 0]], # instance 2 -- zero padded
        [[6, 7, 8], [6, 5, 4]], # instance 3
        [[9, 0, 1], [3, 2, 1]], # instance 4
    ])

seq_length_batch = np.array([2,1,2,2])

init = tf.global_variables_initializer()

with tf.Session() as sess:
    init.run()
    outputs_val, states_val = sess.run(
        [outputs, states],
        feed_dict={X: X_batch, seq_length: seq_length_batch})

# RNN should output zero vectors for any time step
# beyond input sequence length
print(outputs_val)
```

```
[[[ 0.28581977 -0.77421445 -0.34181327 -0.87767971 -0.91387445]
  [ 0.99970448 -1.          0.79238343 -1.         -0.9997654 ]]

 [[ 0.96786171 -0.99937457 -0.03243476 -0.99988878 -0.99875116]
  [ 0.          0.          0.          0.          0.        ]]

 [[ 0.99903995 -0.99999839  0.28328663 -0.99999982 -0.99998271]
  [ 0.96896154 -0.99999189  0.43341497 -0.99996883 -0.98279852]]

 [[ 0.9976812  -0.99999118  0.99979782 -0.99983948  0.84931362]
  [ 0.57188803 -0.99268627 -0.30526906 -0.99518502  0.109933  ]]
]
```

```
# states tensor contains final state of each cell
print(states_val)
```

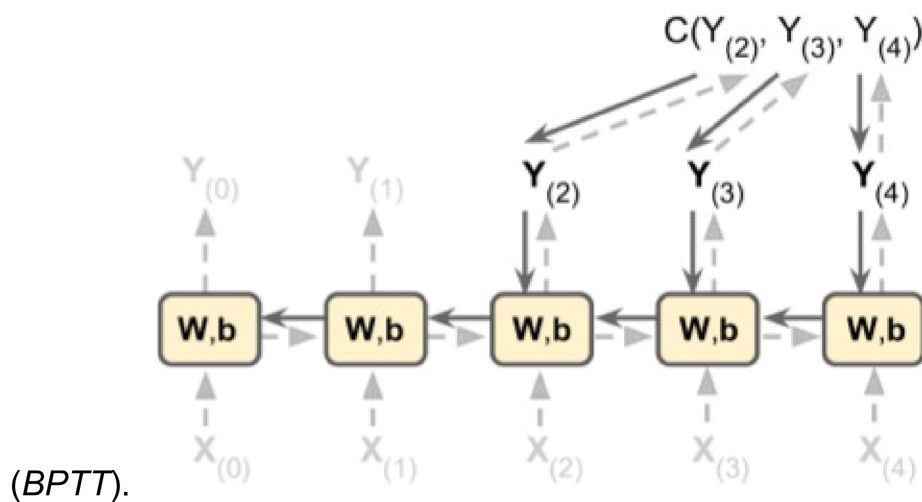
```
[[ 0.99970448 -1.          0.79238343 -1.         -0.9997654 ]
 [ 0.96786171 -0.99937457 -0.03243476 -0.99988878 -0.99875116]
 [ 0.96896154 -0.99999189  0.43341497 -0.99996883 -0.98279852]
 [ 0.57188803 -0.99268627 -0.30526906 -0.99518502  0.109933  ]]
```

Variable-Length Output Sequences

- Typical output sequence lengths not equal to input lengths
- Most common solution: use *end-of-sequence (EOS) token*.

RNN Training

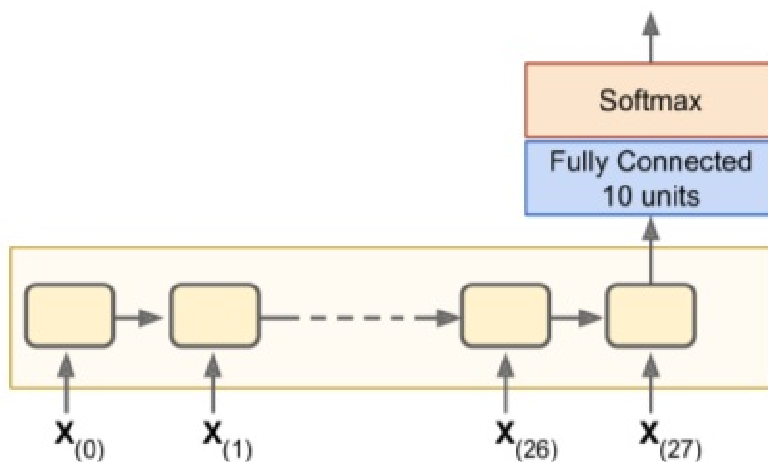
- Unroll through time (as shown above) then use backprop through time



(*BPTT*).

RNN Training: Classifier

- Example: use MNIST (CNN would be better, but lets keep it simple)
- Treat images as 28 rows of 28 pixels each
- Use 150 rnn cells + fully-connected layer of 10 cells (1 per class)
- Followed by softmax layer




```
# similar to MNIST classifier
# unrolled RNN replaces hidden layers

tf.reset_default_graph()

from tensorflow.contrib.layers import fully_connected

n_steps = 28
n_inputs = 28
n_neurons = 150
n_outputs = 10
```

```
learning_rate = 0.001

# y = placeholder for target classes

X = tf.placeholder(tf.float32, [None, n_steps, n_inputs])
y = tf.placeholder(tf.int32, [None])

basic_cell = tf.contrib.rnn.BasicRNNCell(
    num_units=n_neurons)

outputs, states = tf.nn.dynamic_rnn(
    basic_cell, X, dtype=tf.float32)

logits = fully_connected(
    states, n_outputs, activation_fn=None)

xentropy = tf.nn.sparse_softmax_cross_entropy_with_logits(
    labels=y, logits=logits)

loss = tf.reduce_mean(
    xentropy)

optimizer = tf.train.AdamOptimizer(
    learning_rate=learning_rate)

training_op = optimizer.minimize(
    loss)

correct = tf.nn.in_top_k(
    logits, y, 1)

accuracy = tf.reduce_mean(
    tf.cast(correct, tf.float32))

init = tf.global_variables_initializer()
```

```
# load MNIST data, reshape to [batch_size, n_steps, n_inputs]

from tensorflow.examples.tutorials.mnist import input_data

mnist = input_data.read_data_sets("/tmp/data/")

X_test = mnist.test.images.reshape((-1, n_steps, n_inputs))
y_test = mnist.test.labels
```

```
Extracting /tmp/data/train-images-idx3-ubyte.gz
Extracting /tmp/data/train-labels-idx1-ubyte.gz
Extracting /tmp/data/t10k-images-idx3-ubyte.gz
Extracting /tmp/data/t10k-labels-idx1-ubyte.gz
```

```
# ready to run. reshape each training batch before feeding to ne
t.

n_epochs = 10
batch_size = 150

with tf.Session() as sess:
    init.run()
    for epoch in range(n_epochs):
        for iteration in range(mnist.train.num_examples // batch
_size):

            X_batch, y_batch = mnist.train.next_batch(batch_size
)
            X_batch = X_batch.reshape(
                (-1, n_steps, n_inputs))

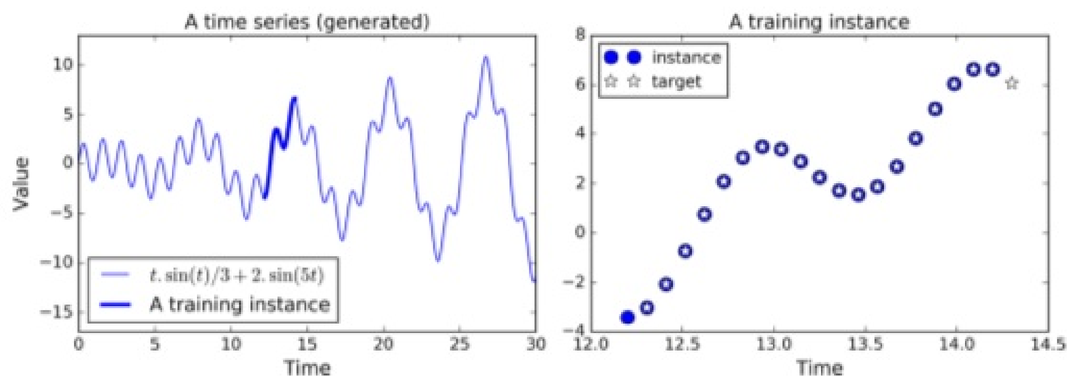
            sess.run(
                training_op,
                feed_dict={X: X_batch, y: y_batch})

        acc_train = accuracy.eval(
            feed_dict={X: X_batch, y: y_batch})
        acc_test = accuracy.eval(
            feed_dict={X: X_test, y: y_test})

        print(epoch,
              "Train accuracy:", acc_train,
              "Test accuracy:",  acc_test)
```

```
0 Train accuracy: 0.953333 Test accuracy: 0.8711
1 Train accuracy: 0.953333 Test accuracy: 0.9417
2 Train accuracy: 0.953333 Test accuracy: 0.9432
3 Train accuracy: 0.946667 Test accuracy: 0.9595
4 Train accuracy: 0.98 Test accuracy: 0.9627
5 Train accuracy: 0.966667 Test accuracy: 0.9666
6 Train accuracy: 0.96 Test accuracy: 0.961
7 Train accuracy: 0.973333 Test accuracy: 0.9729
8 Train accuracy: 0.986667 Test accuracy: 0.9702
9 Train accuracy: 0.986667 Test accuracy: 0.9732
```

RNN Training: Predicting Time Series



```
t_min, t_max = 0, 30
resolution = 0.1

def time_series(t):
    return t * np.sin(t) / 3 + 2 * np.sin(t*5)

def next_batch(batch_size, n_steps):
    t0 = np.random.rand(batch_size, 1) * (t_max - t_min - n_step
s * resolution)
    Ts = t0 + np.arange(0., n_steps + 1) * resolution
    ys = time_series(Ts)
    return ys[:, :-1].reshape(-1, n_steps, 1), ys[:, 1:].reshape(
-1, n_steps, 1)

t = np.linspace(t_min, t_max, (t_max - t_min) // resolution)

n_steps = 20
t_instance = np.linspace(
    12.2, 12.2 + resolution * (n_steps + 1), n_steps + 1)
```

```
# each training instance = 20 inputs long
# targets = 20-input sequences

tf.reset_default_graph()

n_steps = 20
n_inputs = 1
n_neurons = 100
n_outputs = 1

X = tf.placeholder(tf.float32, [None, n_steps, n_inputs])
y = tf.placeholder(tf.float32, [None, n_steps, n_outputs])

cell = tf.contrib.rnn.BasicRNNCell(
    num_units=n_neurons,
    activation=tf.nn.relu)

outputs, states = tf.nn.dynamic_rnn(
    cell, X, dtype=tf.float32)

print(outputs.shape)
```

```
(?, 20, 100)
```

```
# output at each time step now vector[100],
# but we want single output value at each step.

# use OutputProjectionWrapper()
# -- adds FC layer to top of each output

cell = tf.contrib.rnn.OutputProjectionWrapper(
    tf.contrib.rnn.BasicRNNCell(
        num_units=n_neurons,
        activation=tf.nn.relu),
    output_size=n_outputs)
```

```
# define cost function using MSE
# use Adam optimizer

learning_rate = 0.001
loss = tf.reduce_mean(
    tf.square(outputs - y))

optimizer = tf.train.AdamOptimizer(
    learning_rate=learning_rate)

training_op = optimizer.minimize(loss)
init = tf.global_variables_initializer()
```

```
# initialize & run

init = tf.global_variables_initializer()
n_iterations = 1000
batch_size = 50

with tf.Session() as sess:
    init.run()
    for iteration in range(n_iterations):
        X_batch, y_batch = next_batch(batch_size, n_steps)
        sess.run(training_op, feed_dict={X: X_batch, y: y_batch}
)
        if iteration % 100 == 0:
            mse = loss.eval(feed_dict={X: X_batch, y: y_batch})
            print(iteration, "\tMSE:", mse)


    # use trained model to make some predictions
    X_new = time_series(np.array(t_instance[:-1].reshape(-1, n_s
teps, n_inputs)))
    y_pred = sess.run(outputs, feed_dict={X: X_new})
    print(y_pred)
```

```
0     MSE: 15.3099
100     MSE: 13.5276
200     MSE: 11.0956
300     MSE: 9.91156
400     MSE: 14.0311
500     MSE: 9.73811
600     MSE: 9.23351
700     MSE: 9.64445
800     MSE: 8.98904
900     MSE: 10.849
[[[ 0.          0.          0.          ...,  0.          0.
     0.        ]
  [ 0.          0.04218276  0.          ...,  0.          0.
     0.        ]
  [ 0.          0.14342034  0.          ...,  0.          0.
     0.        ]
  ...,
  [ 6.67315388  0.          6.39087296 ...,  6.9017005   6.30435
514
    6.23329258]
  [ 6.61708975  0.          6.31429434 ...,  6.58116341  6.19745
445
    6.11896658]
  [ 5.9406209   0.          5.73649979 ...,  5.63920403  5.53866
72
    5.47510672]]]
```

```
import matplotlib.pyplot as plt

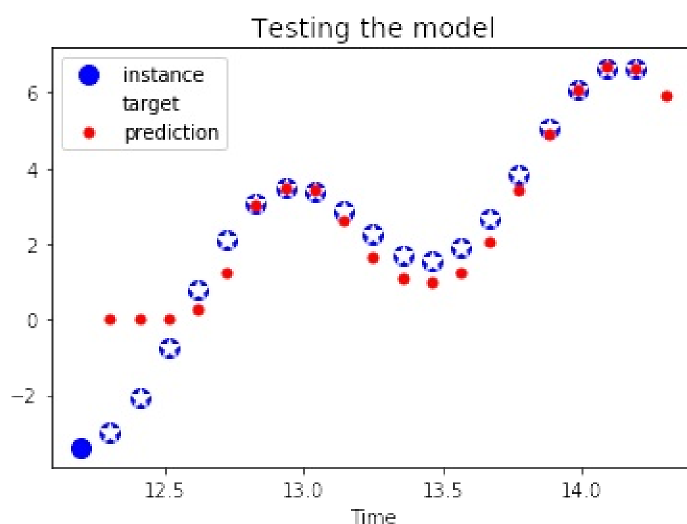
plt.title("Testing the model", fontsize=14)

plt.plot(
    t_instance[:-1],
    time_series(t_instance[:-1]),
    "bo", markersize=10, label="instance")

plt.plot(
    t_instance[1:],
    time_series(t_instance[1:]),
    "w*", markersize=10, label="target")

plt.plot(
    t_instance[1:],
    y_pred[0,:,0],
    "r.", markersize=10, label="prediction")

plt.legend(loc="upper left")
plt.xlabel("Time")
#save_fig("time_series_pred_plot")
plt.show()
```



- **OutputProjectionWrapper()** = simplest solution for reducing output sequences to one value/timestep, but not most efficient.
- More efficient solution shown below - **signficant speed boost**.

```
tf.reset_default_graph()

n_steps = 20
n_inputs = 1
n_neurons = 100
n_outputs = 1

X = tf.placeholder(tf.float32, [None, n_steps, n_inputs])
y = tf.placeholder(tf.float32, [None, n_steps, n_outputs])

cell = tf.contrib.rnn.BasicRNNCell(
    num_units=n_neurons,
    activation=tf.nn.relu)

rnn_outputs, states = tf.nn.dynamic_rnn(
    cell, X, dtype=tf.float32)

# stack outputs using reshape
stacked_rnn_outputs = tf.reshape(
    rnn_outputs, [-1, n_neurons])

print(stacked_rnn_outputs)

# add FC layer -- just a projection, so no activation fn needed
stacked_outputs = fully_connected(
    stacked_rnn_outputs,
    n_outputs,
    activation_fn=None)

print(stacked_outputs)

# unstack outputs using reshape
outputs = tf.reshape(
    stacked_outputs, [-1, n_steps, n_outputs])

print(outputs)

loss = tf.reduce_sum(tf.square(outputs - y))
optimizer = tf.train.AdamOptimizer(learning_rate=learning_rate)
```

```
training_op = optimizer.minimize(loss)

#initialize & run
init = tf.global_variables_initializer()

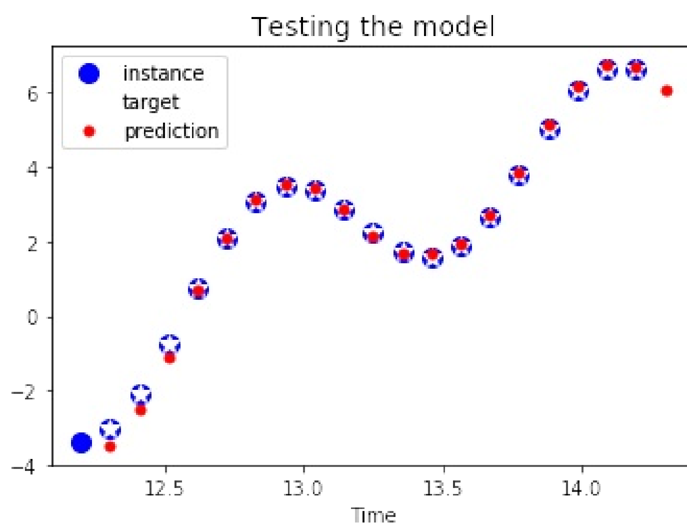
n_iterations = 1000
batch_size = 50

with tf.Session() as sess:
    init.run()
    for iteration in range(n_iterations):
        X_batch, y_batch = next_batch(batch_size, n_steps)
        sess.run(training_op, feed_dict={X: X_batch, y: y_batch}
)
        if iteration % 100 == 0:
            mse = loss.eval(feed_dict={X: X_batch, y: y_batch})
            print(iteration, "\tMSE:", mse)


    # use trained model to make some predictions
    X_new = time_series(np.array(t_instance[:-1].reshape(-1, n_s
teps, n_inputs)))
    y_pred = sess.run(outputs, feed_dict={X: X_new})
    print(y_pred)
```

```
Tensor("Reshape:0", shape=(?, 100), dtype=float32)
Tensor("fully_connected/BiasAdd:0", shape=(?, 1), dtype=float32)
Tensor("Reshape_1:0", shape=(?, 20, 1), dtype=float32)
0     MSE: 22963.7
100     MSE: 743.444
200     MSE: 276.131
300     MSE: 117.955
400     MSE: 53.3529
500     MSE: 63.4189
600     MSE: 45.1415
700     MSE: 41.5129
800     MSE: 53.4219
900     MSE: 43.2203
[[[-3.46527553]
  [-2.46867704]
  [-1.10144436]
  [ 0.69717044]
  [ 2.08823276]
  [ 3.13628578]
  [ 3.55210543]
  [ 3.4186697 ]
  [ 2.85978389]
  [ 2.15520501]
  [ 1.67705297]
  [ 1.6919663 ]
  [ 1.93633199]
  [ 2.70151305]
  [ 3.87054777]
  [ 5.11770582]
  [ 6.15701818]
  [ 6.71814394]
  [ 6.69798708]
  [ 6.08309698]]]
```

```
plt.title("Testing the model", fontsize=14)
plt.plot(t_instance[:-1], time_series(t_instance[:-1]), "bo", ma
rkersize=10, label="instance")
plt.plot(t_instance[1:], time_series(t_instance[1:]), "w*", mark
ersize=10, label="target")
plt.plot(t_instance[1:], y_pred[0,:,0], "r.", markersize=10, lab
el="prediction")
plt.legend(loc="upper left")
plt.xlabel("Time")
plt.show()
```



Creative RNNs

- Use model to generate creative sequences
- Provide seed sequence of length = n_steps, zero-filled
- use model to append predicted new value to sequence
- feed last n_steps values to model to predict next value, etc.
- should get new sequence resembling original time series

```
n_iterations = 2000
batch_size = 50

with tf.Session() as sess:
    init.run()
    for iteration in range(n_iterations):
        X_batch, y_batch = next_batch(batch_size, n_steps)
```

```
        sess.run(training_op, feed_dict={X: X_batch, y: y_batch}
)
        if iteration % 100 == 0:
            mse = loss.eval(feed_dict={X: X_batch, y: y_batch})
            print(iteration, "\tMSE:", mse)

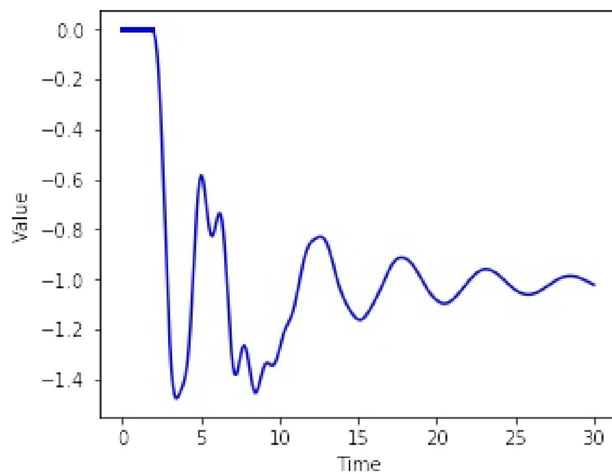
    sequence1 = [0. for i in range(n_steps)]
    for iteration in range(len(t) - n_steps):
        X_batch = np.array(sequence1[-n_steps:]).reshape(1, n_st
eps, 1)
        y_pred = sess.run(outputs, feed_dict={X: X_batch})
        sequence1.append(y_pred[0, -1, 0])

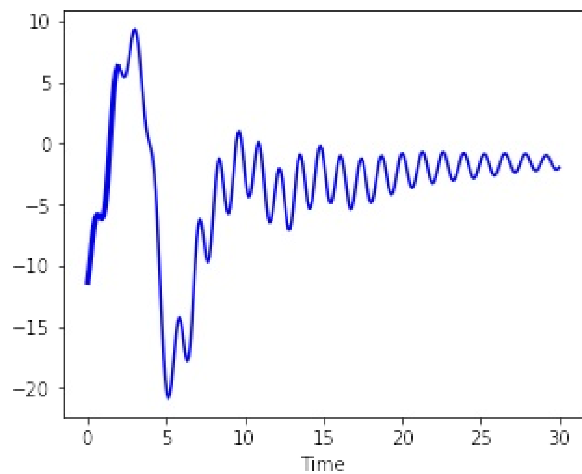
    sequence2 = [time_series(i * resolution + t_min + (t_max-t_m
in/3)) for i in range(n_steps)]
    for iteration in range(len(t) - n_steps):
        X_batch = np.array(sequence2[-n_steps:]).reshape(1, n_st
eps, 1)
        y_pred = sess.run(outputs, feed_dict={X: X_batch})
        sequence2.append(y_pred[0, -1, 0])

plt.figure(figsize=(11,4))
plt.subplot(121)
plt.plot(t, sequence1, "b-")
plt.plot(t[:n_steps], sequence1[:n_steps], "b-", linewidth=3)
plt.xlabel("Time")
plt.ylabel("Value")

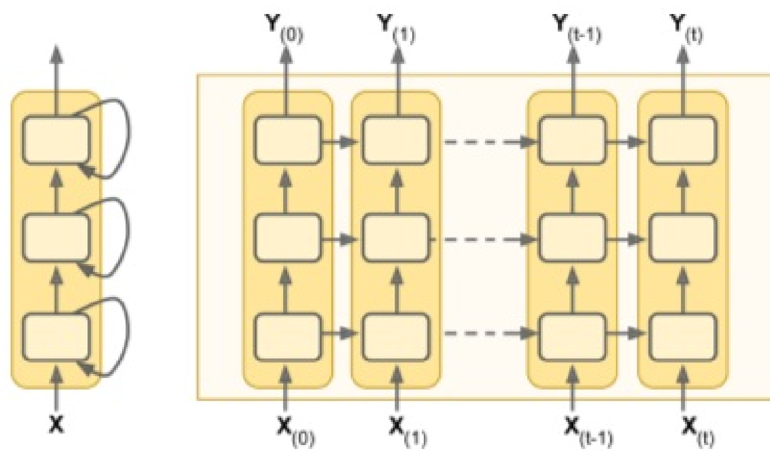
plt.subplot(122)
plt.plot(t, sequence2, "b-")
plt.plot(t[:n_steps], sequence2[:n_steps], "b-", linewidth=3)
plt.xlabel("Time")
#save_fig("creative_sequence_plot")
plt.show()
```

```
0     MSE: 14607.1
100     MSE: 505.605
200     MSE: 167.29
300     MSE: 83.1336
400     MSE: 58.9695
500     MSE: 61.0224
600     MSE: 55.8671
700     MSE: 43.7078
800     MSE: 57.2013
900     MSE: 55.3992
1000     MSE: 54.082
1100     MSE: 55.48
1200     MSE: 39.4618
1300     MSE: 40.7414
1400     MSE: 47.8548
1500     MSE: 43.9252
1600     MSE: 47.892
1700     MSE: 42.0762
1800     MSE: 48.2429
1900     MSE: 42.7509
```





Deep RNNs



- Built by stacking cells into a *MultiRNNCell()*.

```
tf.reset_default_graph()

n_inputs = 2
n_neurons = 100
n_layers = 3
n_steps = 5
keep_prob = 0.5

X = tf.placeholder(tf.float32, [None, n_steps, n_inputs])

basic_cell = tf.contrib.rnn.BasicRNNCell(
    num_units=n_neurons)

print(basic_cell)

multi_layer_cell = tf.contrib.rnn.MultiRNNCell(
    [basic_cell] * n_layers)

print(multi_layer_cell)

# states = tuple (one tensor/layer, = final state of layer's cel
l)

outputs, states = tf.nn.dynamic_rnn(
    multi_layer_cell, X, dtype=tf.float32)

init = tf.global_variables_initializer()

import numpy.random as rnd
X_batch = rnd.rand(2, n_steps, n_inputs)

with tf.Session() as sess:
    init.run()
    outputs_val, states_val = sess.run(
        [outputs, states],
        feed_dict={X: X_batch})

print(outputs_val.shape)
```

```
<tensorflow.contrib.rnn.python.ops.core_rnn_cell_impl.BasicRNNCe
ll object at 0x7fd1ff3dbb00>
<tensorflow.contrib.rnn.python.ops.core_rnn_cell_impl.MultiRNNCe
ll object at 0x7fd1d9b7c9e8>
(2, 5, 100)
```

DRNNs: Multiple GPUs

- **TO DO**

Dropout

- Very deep RNNs = danger of overfit. Use dropout to avoid problem.
- Can apply before or after RNN
- If applying dropout between RNN layers, need to use *DropoutWrapper*.

```
# apply 50% dropout to inputs of RNN layers
# can apply dropout to outputs via output_keep_prob

tf.reset_default_graph()
from tensorflow.contrib.layers import fully_connected

n_inputs = 1
n_neurons = 100
n_layers = 3
n_steps = 20
n_outputs = 1

keep_prob = 0.5
learning_rate = 0.001

def deep_rnn_with_dropout(X, y, is_training):

    # TF implementation of DropoutWrapper doesn't differentiate
    # between training & testing.

    cell = tf.contrib.rnn.BasicRNNCell(
```

```
        num_units=n_neurons)

    if is_training:
        cell = tf.contrib.rnn.DropoutWrapper(
            cell, input_keep_prob=keep_prob)

    #
    #

    multi_layer_cell = tf.contrib.rnn.MultiRNNCell(
        [cell] * n_layers)

    rnn_outputs, states = tf.nn.dynamic_rnn(
        multi_layer_cell, X, dtype=tf.float32)

    stacked_rnn_outputs = tf.reshape(
        rnn_outputs, [-1, n_neurons])

    stacked_outputs = fully_connected(
        stacked_rnn_outputs, n_outputs, activation_fn=None)

    outputs = tf.reshape(
        stacked_outputs, [-1, n_steps, n_outputs])

    loss = tf.reduce_sum(
        tf.square(outputs - y))

    optimizer = tf.train.AdamOptimizer(
        learning_rate=learning_rate)

    training_op = optimizer.minimize(loss)

    return outputs, loss, training_op

X = tf.placeholder(tf.float32, [None, n_steps, n_inputs])
y = tf.placeholder(tf.float32, [None, n_steps, n_outputs])
outputs, loss, training_op = deep_rnn_with_dropout(X, y, is_trai
ning)
init = tf.global_variables_initializer()
saver = tf.train.Saver()
```

- Dropout, in this code, works during both training & testing (don't want).
- *dropout_wrapper()* doesn't know how to handle this, so you need one graph for training, another for testing.

```
n_iterations = 2000
batch_size = 50

is_training = True

with tf.Session() as sess:
    if is_training:
        init.run()
        for iteration in range(n_iterations):
            X_batch, y_batch = next_batch(batch_size, n_steps)
            sess.run(
                training_op,
                feed_dict={X: X_batch, y: y_batch})

            if iteration % 100 == 0:
                mse = loss.eval(
                    feed_dict={X: X_batch, y: y_batch})

                print(iteration, "\tMSE:", mse)

        save_path = saver.save(sess, "/tmp/my_model.ckpt")

    else:
        saver.restore(sess, "/tmp/my_model.ckpt")

        X_new = time_series(
            np.array(t_instance[:-1].reshape(-1, n_steps, n_inpu
ts)))
        y_pred = sess.run(
            outputs, feed_dict={X: X_new})

        plt.title("Testing the model", fontsize=14)
        plt.plot(t_instance[:-1], time_series(t_instance[:-1]),
"bo", markersize=10, label="instance")
        plt.plot(t_instance[1:], time_series(t_instance[1:]), "w
```

```
*", markersize=10, label="target")
        plt.plot(t_instance[1:], y_pred[0,:,0], "r.", markersize=
10, label="prediction")
        plt.legend(loc="upper left")
        plt.xlabel("Time")
        plt.show()
```

```
0     MSE: 10428.8
100     MSE: 314.521
200     MSE: 152.328
300     MSE: 155.774
400     MSE: 100.226
500     MSE: 80.2064
600     MSE: 92.3898
700     MSE: 55.4301
800     MSE: 50.8537
900     MSE: 47.1413
1000     MSE: 57.1007
1100     MSE: 64.2314
1200     MSE: 51.3272
1300     MSE: 51.1612
1400     MSE: 41.0518
1500     MSE: 42.267
1600     MSE: 29.6838
1700     MSE: 48.4316
1800     MSE: 46.5584
1900     MSE: 40.6252
```

```
# testing

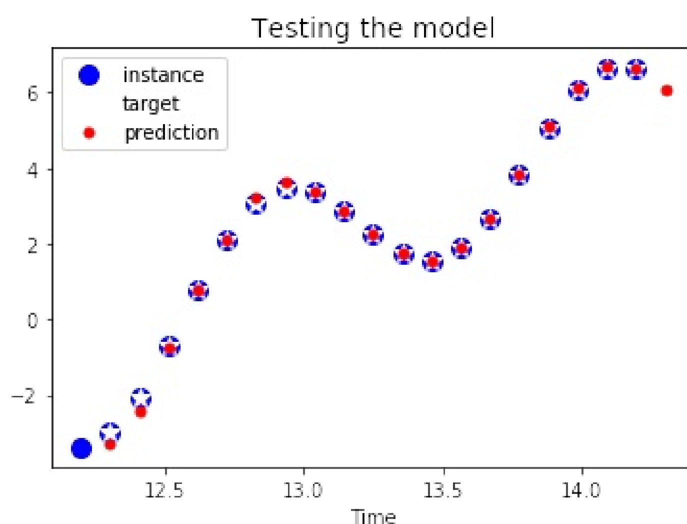
with tf.Session() as sess:

    saver.restore(sess, "/tmp/my_model.ckpt")

    X_new = time_series(
        np.array(t_instance[:-1].reshape(-1, n_steps, n_inputs))
)

    y_pred = sess.run(
        outputs, feed_dict={X: X_new})

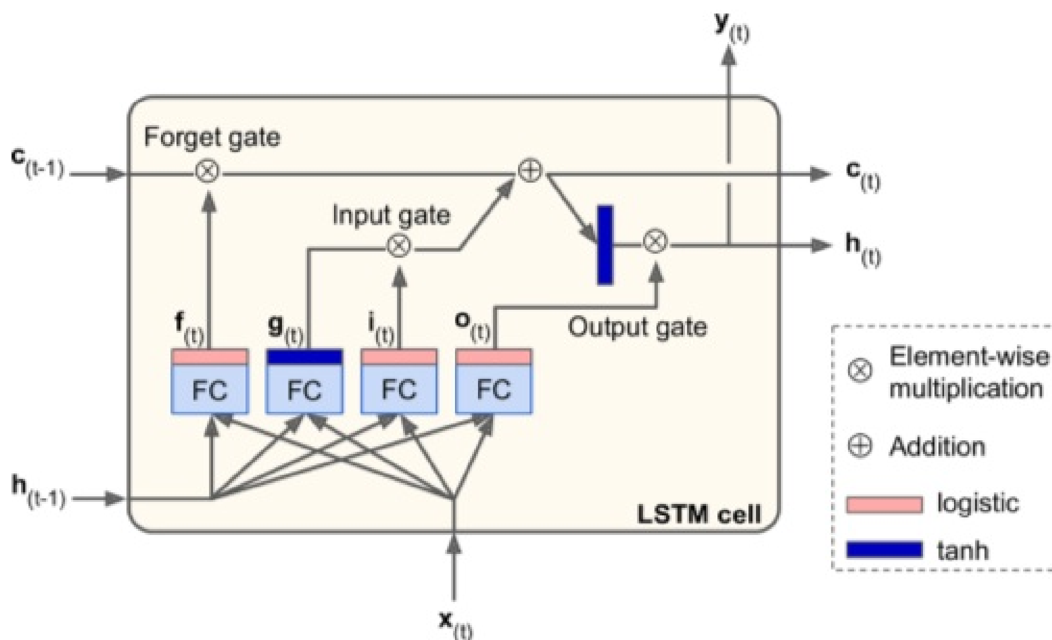
    plt.title("Testing the model", fontsize=14)
    plt.plot(t_instance[:-1], time_series(t_instance[:-1]), "bo"
, markersize=10, label="instance")
    plt.plot(t_instance[1:], time_series(t_instance[1:]), "w*",
markersize=10, label="target")
    plt.plot(t_instance[1:], y_pred[0,:,0], "r.", markersize=10,
 label="prediction")
    plt.legend(loc="upper left")
    plt.xlabel("Time")
    plt.show()
```



Training across Many Time Steps

- problem #1: RNNs susceptible to vanishing/exploding gradients issues. Previous tricks will work, but training time = prohibitively long for even modest sequences.
- solution #1: *truncated backprop thru time* (unrolling RNN over limited number of timesteps during training). Works, but *model will not be able to learn long-term patterns*.
- problem #2: memory of early inputs fades away - information lost during each transformation.
- solution #2: using a *long-term memory cell*.

Long Short-Term Memory (LSTM) Cell



- implemented via *BasicLSTMCell()* instead of *BasicRNNCell()*.

- key feature: net learns what to store (long-term), what to read from, what to throw away.

- Four FC layers - each with unique purposes:
  - main layer: outputs g(t)
  - forget gate: controlled by f(t) - decides which parts of long-term memory to erase
  - input gate: controlled by i(t) - decides which parts of g(t) to add to long-term memory
  - output gate: controlled by o(t) - decides which parts of long-term state

should be read & outputted at this time step.

```
tf.reset_default_graph()

from tensorflow.contrib.layers import fully_connected

n_steps = 28
n_inputs = 28
n_neurons = 150
n_outputs = 10

learning_rate = 0.001

X = tf.placeholder(tf.float32, [None, n_steps, n_inputs])
y = tf.placeholder(tf.int32, [None])

lstm_cell = tf.contrib.rnn.BasicLSTMCell(
    num_units=n_neurons)

multi_cell = tf.contrib.rnn.MultiRNNCell(
    [lstm_cell]*3)

outputs, states = tf.nn.dynamic_rnn(
    multi_cell, X, dtype=tf.float32)

top_layer_h_state = states[-1][1]

logits = fully_connected(
    top_layer_h_state,
    n_outputs,
    activation_fn=None, scope="softmax")

xentropy = tf.nn.sparse_softmax_cross_entropy_with_logits(
    labels=y, logits=logits)

loss = tf.reduce_mean(
    xentropy, name="loss")

optimizer = tf.train.AdamOptimizer(
    learning_rate=learning_rate)
```

```
training_op = optimizer.minimize(loss)

correct = tf.nn.in_top_k(
    logits, y, 1)

accuracy = tf.reduce_mean(
    tf.cast(correct, tf.float32))

init = tf.global_variables_initializer()
```

```
states
```

```
(LSTMStateTuple(c=<tf.Tensor 'rnn/while/Exit_2:0' shape=(?, 150)
 dtype=float32>, h=<tf.Tensor 'rnn/while/Exit_3:0' shape=(?, 150
) dtype=float32>),
 LSTMStateTuple(c=<tf.Tensor 'rnn/while/Exit_4:0' shape=(?, 150)
 dtype=float32>, h=<tf.Tensor 'rnn/while/Exit_5:0' shape=(?, 150
) dtype=float32>),
 LSTMStateTuple(c=<tf.Tensor 'rnn/while/Exit_6:0' shape=(?, 150)
 dtype=float32>, h=<tf.Tensor 'rnn/while/Exit_7:0' shape=(?, 150
) dtype=float32>))
```

```
top_layer_h_state
```

```
<tf.Tensor 'rnn/while/Exit_7:0' shape=(?, 150) dtype=float32>
```

```
n_epochs = 10
batch_size = 150

with tf.Session() as sess:
    init.run()
    for epoch in range(n_epochs):
        for iteration in range(mnist.train.num_examples // batch
_size):
            X_batch, y_batch = mnist.train.next_batch(batch_size
)
            X_batch = X_batch.reshape((batch_size, n_steps, n_in
puts))
            sess.run(training_op, feed_dict={X: X_batch, y: y_ba
tch})
        acc_train = accuracy.eval(feed_dict={X: X_batch, y: y_ba
tch})
        acc_test = accuracy.eval(feed_dict={X: X_test, y: y_test
})
        print("Epoch", epoch, "Train accuracy =", acc_train, "Te
st accuracy =", acc_test)
```

```
Epoch 0 Train accuracy = 0.966667 Test accuracy = 0.9403
Epoch 1 Train accuracy = 0.98 Test accuracy = 0.9742
Epoch 2 Train accuracy = 0.993333 Test accuracy = 0.979
Epoch 3 Train accuracy = 0.993333 Test accuracy = 0.9805
Epoch 4 Train accuracy = 1.0 Test accuracy = 0.9854
Epoch 5 Train accuracy = 0.98 Test accuracy = 0.9827
Epoch 6 Train accuracy = 0.993333 Test accuracy = 0.9851
Epoch 7 Train accuracy = 1.0 Test accuracy = 0.9865
Epoch 8 Train accuracy = 1.0 Test accuracy = 0.9887
Epoch 9 Train accuracy = 0.993333 Test accuracy = 0.9871
```

Peephole Connections

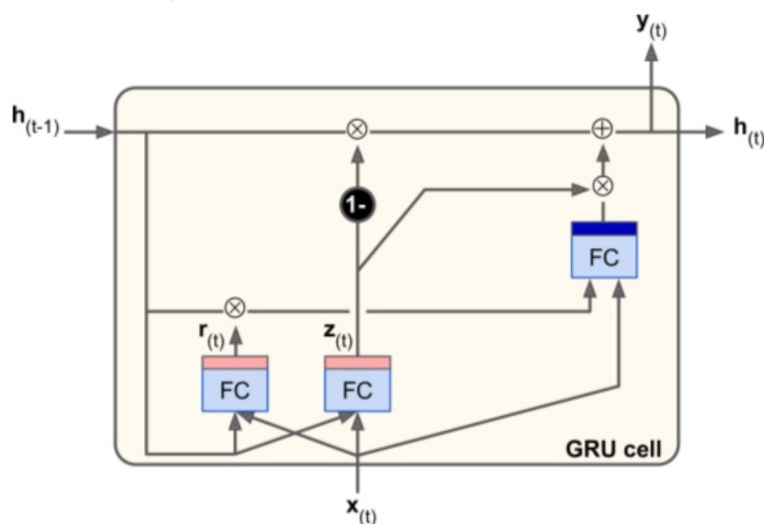
- Basic LSTM cell: gate controllers only see input x(t) & prev short-term state h(t-1).
- Improvement: let gate peek at long-term state too. Provided with previous

long-term state c(t-1) as inputs to forget gate & input gate; current long-term state c(t) added as input to output gate controller.

```
# Peepholes in TF
lstm_cell = tf.contrib.rnn.LSTMCell(
    num_units=n_neurons,
    use_peepholes=True)
```

Gated Recurrent Unit (GRU) Cell

- Simplified version of LSTM cell
- State vectors merged into single h(t).
- Single gate controller manages forget gate & input gate. (if a memory is to be stored, its location is erased first.)
- No output gate - full state vector output on




```
# in TF
gru_cell = tf.contrib.rnn.GRUCell(num_units=n_neurons)
```

Natural Language Processing (NLP)

- Mostly based on RNNs
- See Word2Vec and Seq2Seq tutorials!
- More: Chris Olah, Sebastian Ruder

Word Embeddings

- First: need a word representation. Similar words should have similar representations.
- Common sol'n: each word in vocab = small, dense vector of embeddings.

```
# create embeddings variable. init with random[-1,+1]

vocabulary_size = 50000
embedding_size = 150
embeddings = tf.Variable(
    tf.random_uniform(
        [vocabulary_size, embedding_size],
        -1.0, 1.0))
```

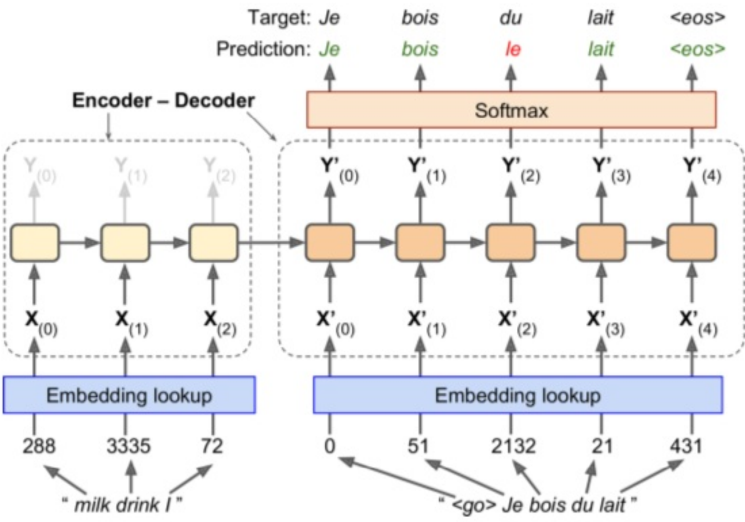
- Feeding new sentences to net: replace unknown words, numbers, URLs, etc with predefined tokens. Once a word is known, you can look it up in a dictionary.

```
train_inputs = tf.placeholder(
    tf.int32, shape=[None]) # from ids...

embed = tf.nn.embedding_lookup(
    embeddings, train_inputs) # ...to embeddings
```

English => French Encoder-Decoder Network (link)

- English inputs, French outputs
- French translations also fed, pushed back one step
- English sentences **reversed** before entry (ensures beginning of sentence is fed last = best for decoder translation)
- Decoder returns score for each word in output vocabulary - softmax turns them into probabilities. Highest probability word is returned.


Target: Je bois du lait <eos>
Prediction: Je bois le lait <eos>
Encoder – Decoder
Softmax
Y(0) Y(1) Y(2)
Y'(0) Y'(1) Y'(2) Y'(3) Y'(4)
X(0) X(1) X(2)
X'(0) X'(1) X'(2) X'(3) X'(4)
Embedding lookup
Embedding lookup
288 3335 72
0 51 2132 21 431
" milk drink I "
" <go> Je bois du lait "

```
from six.moves import urllib

import errno
import os
import zipfile

WORDS_PATH = "datasets/words"
WORDS_URL = 'http://mattmahoney.net/dc/text8.zip'

def mkdir_p(path):
    """Create directories, ok if they already exist.

    This is for python 2 support. In python >=3.2, simply use:
    >>> os.makedirs(path, exist_ok=True)
    """
    try:
        os.makedirs(path)
    except OSError as exc:
        if exc.errno == errno.EEXIST and os.path.isdir(path):
            pass
        else:
            raise

def fetch_words_data(words_url=WORDS_URL, words_path=WORDS_PATH):

    os.makedirs(words_path, exist_ok=True)
    zip_path = os.path.join(words_path, "words.zip")
    if not os.path.exists(zip_path):
        urllib.request.urlretrieve(words_url, zip_path)
    with zipfile.ZipFile(zip_path) as f:
        data = f.read(f.namelist()[0])
    return data.decode("ascii").split()
```

```
words = fetch_words_data()
words[:5]
```

```
['anarchism', 'originated', 'as', 'a', 'term']
```

Build dictionary

```
from collections import Counter

vocabulary_size = 50000

vocabulary = [("UNK", None)] + Counter(words).most_common(vocabu
lary_size - 1)
vocabulary = np.array([word for word, _ in vocabulary])

dictionary = {word: code for code, word in enumerate(vocabulary)
}

data = np.array([dictionary.get(word, 0) for word in words])
```

```
" ".join(words[:9]), data[:9]
```

```
('anarchism originated as a term of abuse first used',
 array([5244, 3081,   12,    6,  195,    2, 3135,   46,   59]))
```

```
" ".join([vocabulary[word_index] for word_index in [5241, 3081,
12, 6, 195, 2, 3134, 46, 59]])
```

```
'anywhere originated as a term of presidency first used'
```

```
words[24], data[24]
```

```
('culottes', 0)
```

Generate batches

```
import random
from collections import deque

def generate_batch(batch_size, num_skips, skip_window):
    global data_index
    assert batch_size % num_skips == 0
    assert num_skips <= 2 * skip_window
    batch = np.ndarray(shape=(batch_size), dtype=np.int32)
    labels = np.ndarray(shape=(batch_size, 1), dtype=np.int32)
    span = 2 * skip_window + 1 # [ skip_window target skip_window ]
    buffer = deque(maxlen=span)
    for _ in range(span):
        buffer.append(data[data_index])
        data_index = (data_index + 1) % len(data)
    for i in range(batch_size // num_skips):
        target = skip_window  # target label at the center of the buffer
        targets_to_avoid = [ skip_window ]
        for j in range(num_skips):
            while target in targets_to_avoid:
                target = random.randint(0, span - 1)
            targets_to_avoid.append(target)
            batch[i * num_skips + j] = buffer[skip_window]
            labels[i * num_skips + j, 0] = buffer[target]
        buffer.append(data[data_index])
        data_index = (data_index + 1) % len(data)
    return batch, labels
```

```
data_index=0
batch, labels = generate_batch(8, 2, 1)
```

```
batch, [vocabulary[word] for word in batch]
```

```
(array([3081, 3081,   12,   12,    6,    6,  195,  195], dtype=i
nt32),
 ['originated', 'originated', 'as', 'as', 'a', 'a', 'term', 'ter
m'])
```

```
labels, [vocabulary[word] for word in labels[:, 0]]
```

```
(array([[5244],
        [  12],
        [   6],
        [3081],
        [ 195],
        [  12],
        [   6],
        [   2]], dtype=int32),
 ['anarchism', 'as', 'a', 'originated', 'term', 'as', 'a', 'of']
)
```

Build the Model

```
batch_size = 128
embedding_size = 128  # Dimension of the embedding vector.
skip_window = 1       # How many words to consider left and right.
num_skips = 2         # How many times to reuse an input to generate a label.

# We pick a random validation set to sample nearest neighbors. Here we limit the
# validation samples to the words that have a low numeric ID, which by
# construction are also the most frequent.

valid_size = 16     # Random set of words to evaluate similarity on.
valid_window = 100  # Only pick dev samples in the head of the distribution.
valid_examples = rnd.choice(valid_window, valid_size, replace=False)
num_sampled = 64    # Number of negative examples to sample.

learning_rate = 0.01
```

```
tf.reset_default_graph()

# Input data.
train_inputs = tf.placeholder(tf.int32, shape=[batch_size])
train_labels = tf.placeholder(tf.int32, shape=[batch_size, 1])
valid_dataset = tf.constant(valid_examples, dtype=tf.int32)

# Look up embeddings for inputs.
init_embeddings = tf.random_uniform([vocabulary_size, embedding_size], -1.0, 1.0)
embeddings = tf.Variable(init_embeddings)
embed = tf.nn.embedding_lookup(embeddings, train_inputs)

# Construct the variables for the NCE loss
nce_weights = tf.Variable(
```

```
    tf.truncated_normal([vocabulary_size, embedding_size],
                        stddev=1.0 / np.sqrt(embedding_size)))
nce_biases = tf.Variable(tf.zeros([vocabulary_size]))

# Compute the average NCE loss for the batch.
# tf.nce_loss automatically draws a new sample of the negative l
abels each
# time we evaluate the loss.
loss = tf.reduce_mean(
    tf.nn.nce_loss(nce_weights, nce_biases, train_labels, embed,
                   num_sampled, vocabulary_size))

# Construct the Adam optimizer
optimizer = tf.train.AdamOptimizer(learning_rate)
training_op = optimizer.minimize(loss)

# Compute the cosine similarity between minibatch examples and a
ll embeddings.
norm = tf.sqrt(tf.reduce_sum(tf.square(embeddings), axis=1, keep
_dims=True))
normalized_embeddings = embeddings / norm
valid_embeddings = tf.nn.embedding_lookup(normalized_embeddings,
 valid_dataset)
similarity = tf.matmul(valid_embeddings, normalized_embeddings,
transpose_b=True)

# Add variable initializer.
init = tf.global_variables_initializer()
```

```
num_steps = 1000 # was 100000?

with tf.Session() as session:
    init.run()

    average_loss = 0
    for step in range(num_steps):
        print("\rIteration: {}".format(step), end="\t")
        batch_inputs, batch_labels = generate_batch(batch_size,
num_skips, skip_window)
```

```
        feed_dict = {train_inputs : batch_inputs, train_labels :
 batch_labels}

        # We perform one update step by evaluating the training
op (including it
        # in the list of returned values for session.run()
        _, loss_val = session.run([training_op, loss], feed_dict
=feed_dict)
        average_loss += loss_val

        if step % 2000 == 0:
            if step > 0:
                average_loss /= 2000
            # The average loss is an estimate of the loss over t
he last 2000 batches.
            print("Average loss at step ", step, ": ", average_l
oss)
            average_loss = 0

        # Note that this is expensive (~20% slowdown if computed
 every 500 steps)
        if step % 10000 == 0:
            sim = similarity.eval()
            for i in range(valid_size):
                valid_word = vocabulary[valid_examples[i]]
                top_k = 8 # number of nearest neighbors
                nearest = (-sim[i, :]).argsort()[1:top_k+1]
                log_str = "Nearest to %s:" % valid_word
                for k in range(top_k):
                    close_word = vocabulary[nearest[k]]
                    log_str = "%s %s," % (log_str, close_word)
                print(log_str)

    final_embeddings = normalized_embeddings.eval()
```

```
Iteration: 0    Average loss at step  0 :  260.603485107
Nearest to and: marsh, sipe, vehement, exercises, einer, mrnas,
dancer, grendel,
Nearest to called: innuendo, algerian, synthesizing, montgomery,
 unspoken, elevating, plankton, monochromatic,
Nearest to many: salinas, fuji, trochaic, rubinstein, eln, tinti
n, lloyd, carbides,
Nearest to about: moreover, congo, choctaws, accomplished, unwie
ldy, ks, halifax, pac,
Nearest to than: awake, exact, offutt, gloster, pronunciations,
delight, tsarina, hopped,
Nearest to or: long, mage, warriors, adhering, sk, clitoridectom
y, parenting, vanguard,
Nearest to of: shakespeare, kemp, relax, cul, breakaway, solemnl
y, mason, mng,
Nearest to when: tolstoy, courtesan, hashes, coursing, evi, ren,
 diurnal, stimson,
Nearest to four: supermassive, soviet, palatalization, acclaimed
, aided, whitney, filtration, lesbians,
Nearest to most: din, hawaii, loch, necronomicon, sunnah, sh, on
ager, miracles,
Nearest to on: helpers, tangle, heretical, compulsion, unorganiz
ed, rump, intimidating, israeli,
Nearest to but: ohio, rican, politeness, watkins, ingesting, str
eet, hatred, novices,
Nearest to that: xhosa, distressed, continually, fausto, iole, a
dmitted, etsi, gross,
Nearest to all: orissa, persistent, moro, informative, reservati
on, ren, browne, frobenius,
Nearest to in: chanced, accelerator, sergio, demonstrating, iner
tia, jarrett, intricate, orange,
Nearest to had: irredentist, kbit, sarris, lactate, bettor, narr
atives, hui, transpired,
Iteration: 999
```

Save final embeddings

```
np.save("my_final_embeddings.npy", final_embeddings)
```

Plot embeddings

```
def plot_with_labels(low_dim_embs, labels):
    assert low_dim_embs.shape[0] >= len(labels), "More labels th
an embeddings"
    plt.figure(figsize=(18, 18))  #in inches
    for i, label in enumerate(labels):
        x, y = low_dim_embs[i,:]
        plt.scatter(x, y)
        plt.annotate(label,
                     xy=(x, y),
                     xytext=(5, 2),
                     textcoords='offset points',
                     ha='right',
                     va='bottom')
    plt.show()
```

```
from sklearn.manifold import TSNE

tsne = TSNE(perplexity=30, n_components=2, init='pca', n_iter=50
00)
plot_only = 500
low_dim_embs = tsne.fit_transform(final_embeddings[:plot_only,:]
)
labels = [vocabulary[i] for i in range(plot_only)]
plot_with_labels(low_dim_embs, labels)
```

```
import matplotlib.pyplot as plt
import numpy as np
import numpy.random as rnd
import tensorflow as tf
import sys
```

Data Representations

- Much easier to remember *sequence patterns* than to remember exact lists. First studied as chess game positions (1970s).
- Autoencoder converts inputs to internal shorthand, then returns best-guess similarity. Two parts: *encoder* (recognizer) & *decoder* (generator, aka *reconstructor*).
- Reconstruction loss - penalizes model when reconstructions /= inputs.
- Internal representation = lower dimensionality, so AE is forced to learn most important features in inputs.

PCA with Undercomplete Linear Autoencoder

```
# lets build a 3D dataset

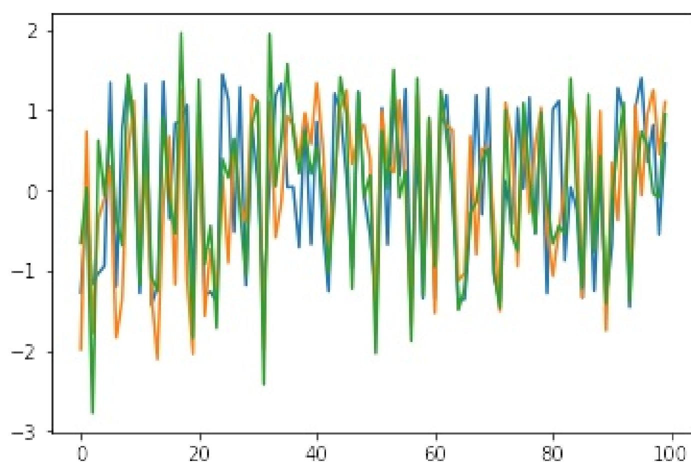
rnd.seed(4)
m = 100
w1, w2 = 0.1, 0.3
noise = 0.1

angles = rnd.rand(m) * 3 * np.pi / 2 - 0.5
X_train = np.empty((m, 3))
X_train[:, 0] = np.cos(angles) + np.sin(angles)/2 + noise * rnd.
randn(m) / 2
X_train[:, 1] = np.sin(angles) * 0.7 + noise * rnd.randn(m) / 2
X_train[:, 2] = X_train[:, 0] * w1 + X_train[:, 1] * w2 + noise
* rnd.randn(m)

# normalize it

from sklearn.preprocessing import StandardScaler
scaler = StandardScaler()
X_train = scaler.fit_transform(X_train)

plt.plot(X_train)
plt.show()
```



```
# build AE

from tensorflow.contrib.layers import fully_connected

n_inputs = 3 # 3D inputs
n_hidden = 2 # 2D codings
n_outputs = n_inputs

learning_rate = 0.01

X = tf.placeholder(
    tf.float32, shape=[None, n_inputs])

#
# set activation_fn=None & use MSE for cost function
# to perform simple PCA.
#

hidden = fully_connected(
    X,
    n_hidden,
    activation_fn=None)

outputs = fully_connected(
    hidden,
    n_outputs,
    activation_fn=None)

# MSE
reconstruction_loss = tf.reduce_mean(
    tf.square(outputs - X))

optimizer = tf.train.AdamOptimizer(
    learning_rate)

training_op = optimizer.minimize(
    reconstruction_loss)

init = tf.global_variables_initializer()
```

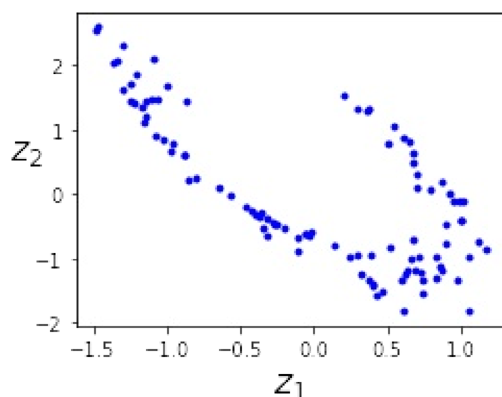
```
# run the AE

n_iterations = 10000
codings = hidden

with tf.Session() as sess:
    init.run()
    for iteration in range(n_iterations):
        training_op.run(feed_dict={X: X_train})
    codings_val = codings.eval(feed_dict={X: X_train})
```

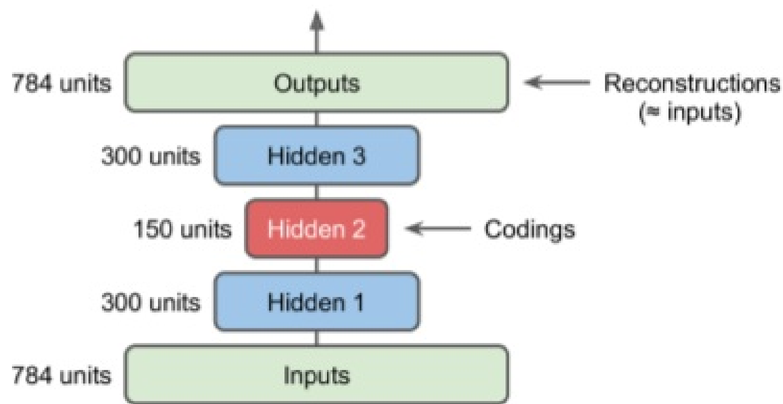
```
fig = plt.figure(figsize=(4,3))
plt.plot(codings_val[:,0], codings_val[:, 1], "b.")
plt.xlabel("$z_1$", fontsize=18)
plt.ylabel("$z_2$", fontsize=18, rotation=0)
#ave_fig("linear_autoencoder_pca_plot")
plt.show()

# plot: 2D projection with max variance
```



Stacked Autoencoders

- AEs with multiple hidden layers - for more complex model learning




```
tf.reset_default_graph()

n_inputs      = 28 * 28 # for MNIST
n_hidden1     = 300
n_hidden2     = 150 # codings
n_hidden3     = n_hidden1
n_outputs     = n_inputs
learning_rate = 0.01
l2_reg        = 0.0001

X = tf.placeholder(tf.float32,
                   shape=[None, n_inputs])

with tf.contrib.framework.arg_scope(
    [fully_connected],
    activation_fn=tf.nn.elu,
    weights_initializer=tf.contrib.layers.variance_scaling_initi
alizer(),
    weights_regularizer=tf.contrib.layers.l2_regularizer(l2_reg)
):

    hidden1 = fully_connected(X,       n_hidden1)
    hidden2 = fully_connected(hidden1, n_hidden2) # codings
    hidden3 = fully_connected(hidden2, n_hidden3)
    outputs = fully_connected(hidden3, n_outputs, activation_fn=
None)

# MSE
reconstruction_loss = tf.reduce_mean(
    tf.square(outputs - X))
```

```
reg_losses = tf.get_collection(
    tf.GraphKeys.REGULARIZATION_LOSSES)

loss = tf.add_n(
    [reconstruction_loss] + reg_losses)

optimizer = tf.train.AdamOptimizer(
    learning_rate)

training_op = optimizer.minimize(loss)

init = tf.global_variables_initializer()
saver = tf.train.Saver()
```

```
# use MNIST dataset

from tensorflow.examples.tutorials.mnist import input_data

mnist = input_data.read_data_sets("/tmp/data/")

# train the net. digit labels (y_batch) = unused.

n_epochs = 4
batch_size = 150

with tf.Session() as sess:
    init.run()
    for epoch in range(n_epochs):
        n_batches = mnist.train.num_examples // batch_size
        for iteration in range(n_batches):
            print("\r{}%".format(100 * iteration // n_batches),
end="")
            sys.stdout.flush()
            X_batch, y_batch = mnist.train.next_batch(batch_size
)
            sess.run(training_op, feed_dict={X: X_batch})
        mse_train = reconstruction_loss.eval(feed_dict={X: X_bat
ch})
        print("\r{}".format(epoch), "Train MSE:", mse_train)
        saver.save(sess, "./my_model_all_layers.ckpt")
```

```
Extracting /tmp/data/train-images-idx3-ubyte.gz
Extracting /tmp/data/train-labels-idx1-ubyte.gz
Extracting /tmp/data/t10k-images-idx3-ubyte.gz
Extracting /tmp/data/t10k-labels-idx1-ubyte.gz
0 Train MSE: 0.02705
1 Train MSE: 0.0137857
2 Train MSE: 0.0113694
3 Train MSE: 0.0107478
```

```
# utility: plot grayscale 28x28 image

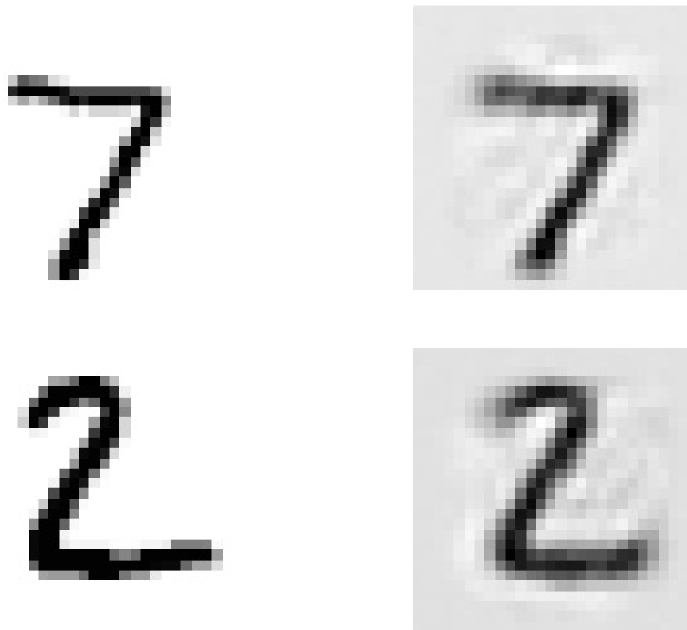
def plot_image(image, shape=[28, 28]):
    plt.imshow(image.reshape(shape), cmap="Greys", interpolation=
"nearest")
    plt.axis("off")
```

```
# load model, eval on test set (measure reconstruction error, di
splay original & reconstruction)

def show_reconstructed_digits(X, outputs, model_path = None, n_t
est_digits = 2):
    with tf.Session() as sess:
        if model_path:
            saver.restore(sess, model_path)
        X_test = mnist.test.images[:n_test_digits]
        outputs_val = outputs.eval(feed_dict={X: X_test})

    fig = plt.figure(figsize=(8, 3 * n_test_digits))
    for digit_index in range(n_test_digits):
        plt.subplot(n_test_digits, 2, digit_index * 2 + 1)
        plot_image(X_test[digit_index])
        plt.subplot(n_test_digits, 2, digit_index * 2 + 2)
        plot_image(outputs_val[digit_index])
```

```
show_reconstructed_digits(X, outputs, "./my_model_all_layers.ckp
t")
plt.show()
```



Tying Weights

- Used when AE is symmetrical. Tying decoder layer weights to encoder layers' weights cuts number of weights by 50% (speedup & less memory).
- Tied weights in TF is cumbersome. Easier to define layers manually.

```
tf.reset_default_graph()

activation = tf.nn.elu

regularizer = tf.contrib.layers.l2_regularizer(l2_reg)

initializer = tf.contrib.layers.variance_scaling_initializer()

X = tf.placeholder(tf.float32, shape=[None, n_inputs])

weights1_init = initializer([n_inputs, n_hidden1])
weights2_init = initializer([n_hidden1, n_hidden2])

weights1 = tf.Variable(weights1_init, dtype=tf.float32, name="we
ights1")
weights2 = tf.Variable(weights2_init, dtype=tf.float32, name="we
ights2")
# weights 3,4 not vars!
```

```
weights3 = tf.transpose(weights2, name="weights3") # tied weights

weights4 = tf.transpose(weights1, name="weights4") # tied weights


biases1 = tf.Variable(tf.zeros(n_hidden1),name="biases1")
biases2 = tf.Variable(tf.zeros(n_hidden2),name="biases2")
biases3 = tf.Variable(tf.zeros(n_hidden3),name="biases3")
biases4 = tf.Variable(tf.zeros(n_outputs),name="biases4")

hidden1 = activation(tf.matmul(X, weights1) + biases1)
hidden2 = activation(tf.matmul(hidden1, weights2) + biases2)
hidden3 = activation(tf.matmul(hidden2, weights3) + biases3)
outputs = tf.matmul(hidden3, weights4) + biases4

reconstruction_loss = tf.reduce_mean(
    tf.square(outputs - X))

reg_loss = regularizer(weights1) + regularizer(weights2)

loss = reconstruction_loss + reg_loss

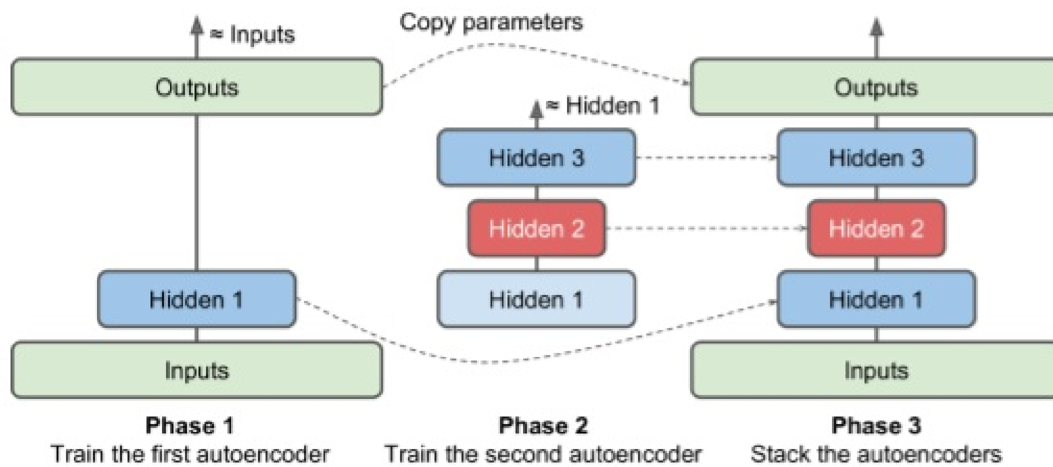
optimizer = tf.train.AdamOptimizer(learning_rate)

training_op = optimizer.minimize(loss)

init = tf.global_variables_initializer()
```

Training one Autoencoder at a time

- Often faster to train each shallow AE individually, then stack them.
- Simplest approach = use separate TF graph for each phase




```
def train_autoencoder(
    X_train,
    n_neurons,
    n_epochs,
    batch_size,
    learning_rate = 0.01,
    l2_reg = 0.0005,
    activation_fn=tf.nn.elu):

    graph = tf.Graph()
    with graph.as_default():
        n_inputs = X_train.shape[1]

        X = tf.placeholder(tf.float32, shape=[None, n_inputs])

        with tf.contrib.framework.arg_scope(
            [fully_connected],
            activation_fn=activation_fn,
            weights_initializer=tf.contrib.layers.variance_scali
ng_initializer(),
            weights_regularizer=tf.contrib.layers.l2_regularizer
(
                l2_reg)):
            hidden = fully_connected(
                X, n_neurons, scope="hidden")
            outputs = fully_connected(
                hidden, n_inputs, activation_fn=None, scope="out
puts")
```

```
        mse = tf.reduce_mean(tf.square(outputs - X))

        reg_losses = tf.get_collection(
            tf.GraphKeys.REGULARIZATION_LOSSES)

        loss = tf.add_n([mse] + reg_losses)

        optimizer = tf.train.AdamOptimizer(learning_rate)

        training_op = optimizer.minimize(loss)

        init = tf.global_variables_initializer()

    with tf.Session(graph=graph) as sess:
        init.run()

        for epoch in range(n_epochs):
            n_batches = len(X_train) // batch_size

            for iteration in range(n_batches):
                print("\r{}%".format(100 * iteration // n_batche
s), end="")

                sys.stdout.flush()

                indices = rnd.permutation(
                    len(X_train))[:batch_size]

                X_batch = X_train[indices]

                sess.run(
                    training_op, feed_dict={X: X_batch})

            mse_train = mse.eval(
                feed_dict={X: X_batch})

            print("\r{}".format(epoch), "Train MSE:", mse_train)

        params = dict(
            [(var.name, var.eval()) for var in tf.get_collection
(
```

```
                tf.GraphKeys.TRAINABLE_VARIABLES)])

        hidden_val = hidden.eval(
            feed_dict={X: X_train})

        return hidden_val, params["hidden/weights:0"], params["hidden/biases:0"], params["outputs/weights:0"], params["outputs/biases:0"]
```

```
# train two AEs

hidden_output, W1, b1, W4, b4 = train_autoencoder(
    mnist.train.images,
    n_neurons=300,
    n_epochs=4,
    batch_size=150)

_, W2, b2, W3, b3 = train_autoencoder(
    hidden_output,
    n_neurons=150,
    n_epochs=4,
    batch_size=150)
```

```
0 Train MSE: 0.0193591
1 Train MSE: 0.0190697
2 Train MSE: 0.0188801
3 Train MSE: 0.0192353
0 Train MSE: 0.00428287
1 Train MSE: 0.00438113
2 Train MSE: 0.00464872
3 Train MSE: 0.00457076
```

```
# create stacked AE by reusing weights &and biases from above

tf.reset_default_graph()

n_inputs = 28*28

X = tf.placeholder(tf.float32, shape=[None, n_inputs])
hidden1 = tf.nn.elu(tf.matmul(X, W1) + b1)
hidden2 = tf.nn.elu(tf.matmul(hidden1, W2) + b2)
hidden3 = tf.nn.elu(tf.matmul(hidden2, W3) + b3)
outputs = tf.matmul(hidden3, W4) + b4
```

Visualizing Reconstructions

```
# Load model, evaluates it on test set (reconstruction error)
# display original & reconstructed images

def show_reconstructed_digits(
    X,
    outputs,
    model_path = None,
    n_test_digits = 2):

    with tf.Session() as sess:
        if model_path:
            saver.restore(sess, model_path)

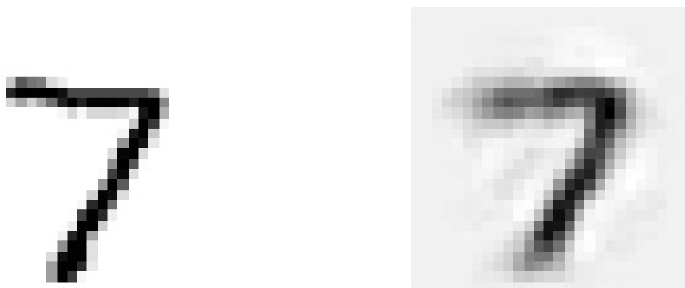
        X_test = mnist.test.images[:n_test_digits]
        outputs_val = outputs.eval(feed_dict={X: X_test})

    fig = plt.figure(figsize=(8, 3 * n_test_digits))

    for digit_index in range(n_test_digits):

        plt.subplot(n_test_digits, 2, digit_index * 2 + 1)
        plot_image(X_test[digit_index])
        plt.subplot(n_test_digits, 2, digit_index * 2 + 2)
        plot_image(outputs_val[digit_index])
        plt.show()

#show_reconstructed_digits(X, outputs, "./my_model_all_layers.ck
pt")
show_reconstructed_digits(X, outputs)
```





Visualizing Features

- simplest method: find training instances that activate each hidden node the most. (best on upper layers, given their tendency to capture high-level features.)

Unsupervised Pretraining with Stacked Autoencoders

Denoising Autoencoders

Sparse Autoencoders

Variational Autoencoders

Other Autoencoders

Intro & Resources

- Sutton/Barto ebook; Silver online course

Learning to Optimize Rewards

- Definitions: software *agents* make *observations* & take *actions* within an *environment*. In return they can receive *rewards* (positive or negative).

Policy Search

- **Policy**: the algorithm used by an agent to determine a next action.

OpenAI Gym (link:)

- A toolkit for various simulated environments.

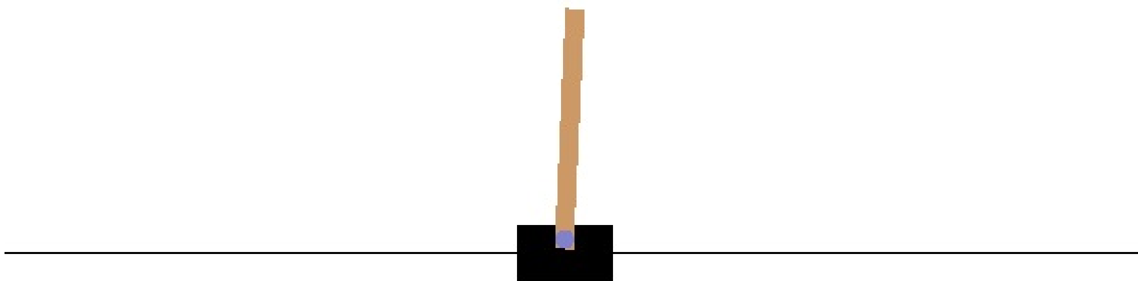
```
!pip3 install --upgrade gym
```

```
Requirement already up-to-date: gym in /home/bjpcjp/anaconda3/li
b/python3.5/site-packages
Requirement already up-to-date: requests>=2.0 in /home/bjpcjp/an
aconda3/lib/python3.5/site-packages (from gym)
Requirement already up-to-date: pyglet>=1.2.0 in /home/bjpcjp/an
aconda3/lib/python3.5/site-packages (from gym)
Requirement already up-to-date: six in /home/bjpcjp/anaconda3/li
b/python3.5/site-packages (from gym)
Requirement already up-to-date: numpy>=1.10.4 in /home/bjpcjp/an
aconda3/lib/python3.5/site-packages (from gym)
```

```
import gym
env = gym.make("CartPole-v0")
obs = env.reset()
obs
env.render()
```

```
[2017-04-27 13:05:47,311] Making new env: CartPole-v0
```

- **make()** creates environment
- **reset()** returns a 1st env't
- **CartPole()** - each observation = 1D numpy array (hposition, velocity, angle, angularvelocity)




```
img = env.render(mode="rgb_array")
img.shape
```

```
(1, 1, 3)
```

```
# what actions are possible?
# in this case: 0 = accelerate left, 1 = accelerate right
env.action_space
```

```
Discrete(2)
```

```
# pole is leaning right. let's go further to the right.
action = 1
obs, reward, done, info = env.step(action)
obs, reward, done, info
```

```
(array([-0.04061536,  0.1486962 , -0.01966318, -0.29249162]), 1.
0, False, {})
```

- new observation:
    - hpos = obs[0]<0
    - velocity = obs[1]>0 = moving to the right
    - angle = obs[2]>0 = leaning right
    - ang velocity = obs[3]<0 = slowing down?
- reward = 1.0
- done = False (episode not over)
- info = (empty)

```
# example policy:
# (1) accelerate left when leaning left, (2) accelerate right wh
en leaning right
# average reward over 500 episodes?

def basic_policy(obs):
    angle = obs[2]
    return 0 if angle < 0 else 1

totals = []
for episode in range(500):
    episode_rewards = 0
    obs = env.reset()
    for step in range(1000): # 1000 steps max, we don't want to
run forever
        action = basic_policy(obs)
        obs, reward, done, info = env.step(action)
        episode_rewards += reward
        if done:
            break
    totals.append(episode_rewards)

import numpy as np
np.mean(totals), np.std(totals), np.min(totals), np.max(totals)
```

```
(41.579999999999998, 8.5249985337242151, 25.0, 62.0)
```

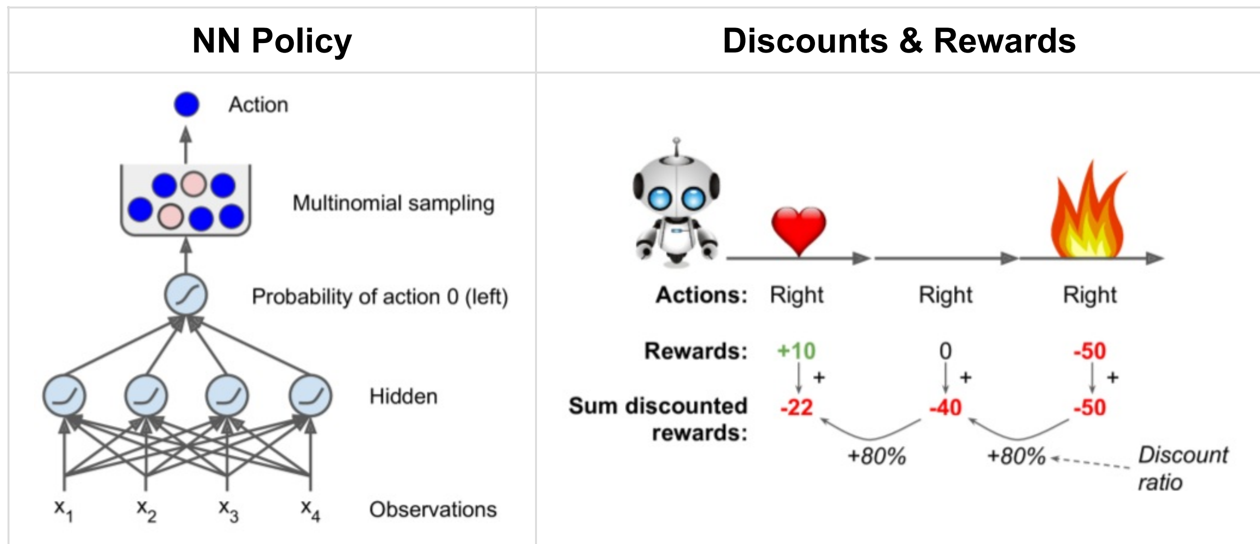
NN Policies

- observations as inputs - actions to be executed as outputs - determined by p(action)
- approach lets agent find best balance between **exploring new actions** & **reusing known good actions**.

Evaluating Actions: Credit Assignment problem

- Reinforcement Learning (RL) training not like supervised learning.
- RL feedback is via rewards (often sparse & delayed)
- How to determine which previous steps were "good" or "bad"? (aka "*credit assigmnment problem*")
- Common tactic: applying a **discount rate** to older rewards.

- Use normalization across many episodes to increase score reliability.



```
import tensorflow as tf
from tensorflow.contrib.layers import fully_connected

# 1. Specify the neural network architecture
n_inputs = 4                        # == env.observation_spa
ce.shape[0]
n_hidden = 4                        # simple task, don't nee
d more hidden neurons
n_outputs = 1                       # only output prob(accel
erating left)
initializer = tf.contrib.layers.variance_scaling_initializer()

# 2. Build the neural network
X = tf.placeholder(
    tf.float32, shape=[None, n_inputs])

hidden = fully_connected(
    X, n_hidden,
    activation_fn=tf.nn.elu,
    weights_initializer=initializer)

logits = fully_connected(
    hidden, n_outputs,
    activation_fn=None,
    weights_initializer=initializer)

outputs = tf.nn.sigmoid(logits)         # logistic (sigmoid) ==
> return 0.0-1.0

# 3. Select a random action based on the estimated probabilities
p_left_and_right = tf.concat(
    axis=1, values=[outputs, 1 - outputs])

action = tf.multinomial(
    tf.log(p_left_and_right),
    num_samples=1)

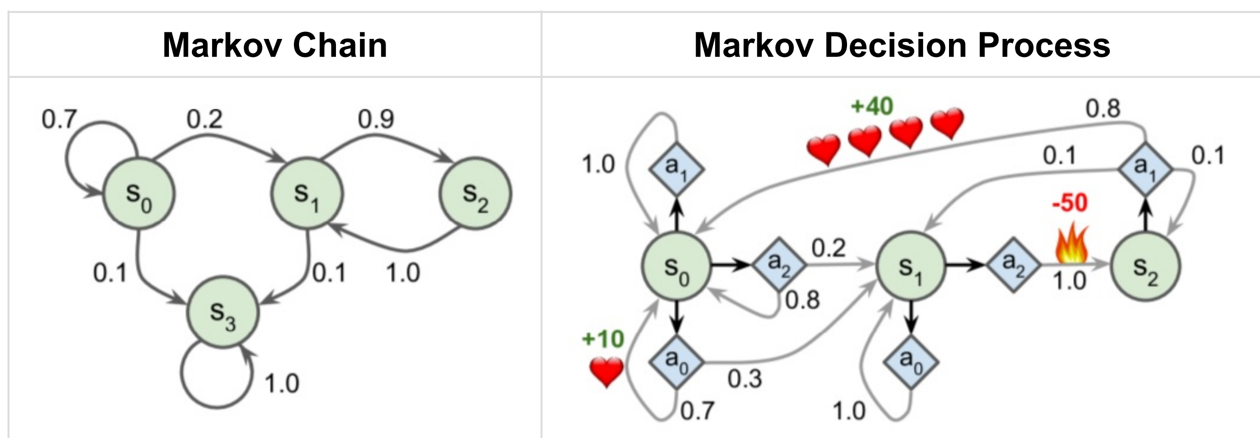
init = tf.global_variables_initializer()
```

Policy Gradient (PG) algorithms

- example: "reinforce" algo, 1992

Markov Decision processes (MDPs)

- Markov chains = stochastic processes, no memory, fixed #states, random transitions
- Markov decision processes = similar to MCs - agent can choose action; transition probabilities depend on the action; transitions can return reward/punishment.
- Goal: find policy with maximum rewards over time.



- **Bellman Optimality Equation**: a method to estimate optimal state value of any state *s*.
- Knowing optimal states = useful, but doesn't tell agent what to do. **Q-Value algorithm** helps solve this problem. Optimal Q-Value of a state-action pair = sum of discounted future rewards the agent can expect on average.

```
# Define MDP:

nan=np.nan # represents impossible actions
T = np.array([ # shape=[s, a, s']
        [[0.7, 0.3, 0.0], [1.0, 0.0, 0.0], [0.8, 0.2, 0.0]],
        [[0.0, 1.0, 0.0], [nan, nan, nan], [0.0, 0.0, 1.0]],
        [[nan, nan, nan], [0.8, 0.1, 0.1], [nan, nan, nan]],
        ])

R = np.array([ # shape=[s, a, s']
```

```
        [[10., 0.0, 0.0], [0.0, 0.0, 0.0], [0.0, 0.0, 0.0]],
        [[10., 0.0, 0.0], [nan, nan, nan], [0.0, 0.0, -50.]],
        [[nan, nan, nan], [40., 0.0, 0.0], [nan, nan, nan]],
        ])

possible_actions = [[0, 1, 2], [0, 2], [1]]

# run Q-Value Iteration algo

Q = np.full((3, 3), -np.inf)
for state, actions in enumerate(possible_actions):
    Q[state, actions] = 0.0 # Initial value = 0.0, for all possi
ble actions

learning_rate = 0.01
discount_rate = 0.95
n_iterations = 100

for iteration in range(n_iterations):
    Q_prev = Q.copy()
    for s in range(3):
        for a in possible_actions[s]:
            Q[s, a] = np.sum([
                T[s, a, sp] * (R[s, a, sp] + discount_rate * np.
max(Q_prev[sp]))
                for sp in range(3)
                ])

print("Q: \n",Q)
print("Optimal action for each state:\n",np.argmax(Q, axis=1))
```

```
Q:
 [[ 21.88646117  20.79149867  16.854807  ]
 [  1.10804034         -inf   1.16703135]
 [        -inf  53.8607061          -inf]]
Optimal action for each state:
 [0 2 1]
```

```
# change discount rate to 0.9, see how policy changes:

discount_rate = 0.90

for iteration in range(n_iterations):
    Q_prev = Q.copy()
    for s in range(3):
        for a in possible_actions[s]:
            Q[s, a] = np.sum([
                T[s, a, sp] * (R[s, a, sp] + discount_rate * np.
max(Q_prev[sp]))
                for sp in range(3)
                ])

print("Q: \n",Q)
print("Optimal action for each state:\n",np.argmax(Q, axis=1))
```

```
Q:
 [[  1.89189499e+01   1.70270580e+01   1.36216526e+01]
 [  3.09979853e-05             -inf  -4.87968388e+00]
 [            -inf   5.01336811e+01             -inf]]
Optimal action for each state:
 [0 0 1]
```

Temporal Difference Learning & Q-Learning

- In general - agent has no knowledge of transition probabilities or rewards
- **Temporal Difference Learning** (TD Learning) similar to value iteration, but accounts for this lack of knowlege.
- Algorithm tracks running average of most recent awards & anticipated rewards.

- **Q-Learning** algorithm adaptation of Q-Value Iteration where initial transition probabilities & rewards are unknown.

```python
import numpy.random as rnd

learning_rate0 = 0.05
learning_rate_decay = 0.1
n_iterations = 20000

s = 0                         # start in state 0
Q = np.full((3, 3), -np.inf)  # -inf for impossible actions

for state, actions in enumerate(possible_actions):
    Q[state, actions] = 0.0 # Initial value = 0.0, for all possible actions
    for iteration in range(n_iterations):
        a = rnd.choice(possible_actions[s]) # choose an action (randomly)
        sp = rnd.choice(range(3), p=T[s, a]) # pick next state using T[s, a]
        reward = R[s, a, sp]

        learning_rate = learning_rate0 / (1 + iteration * learning_rate_decay)

        Q[s, a] = learning_rate * Q[s, a] + (1 - learning_rate) * (reward + discount_rate * np.max(Q[sp]))

s = sp # move to next state

print("Q: \n",Q)
print("Optimal action for each state:\n",np.argmax(Q, axis=1))
```

```
Q: 
 [[             -inf   2.47032823e-323              -inf]
 [  0.00000000e+000              -inf   0.00000000e+000]
 [             -inf   0.00000000e+000              -inf]]
Optimal action for each state:
 [1 0 1]
```

Exploration Policies

- Q-Learning works only if exploration is thorough - not always possible.
- Better alternative: explore more interesting routes using a *sigma* probability

Approximate Q-Learning

- TODO

Ms Pac-Man with Deep Q-Learning

```
env = gym.make('MsPacman-v0')
obs = env.reset()
obs.shape, env.action_space

# action_space = 9 possible joystick actions
# observations = atari screenshots as 3D NumPy arrays
```

```
[2017-04-27 13:06:21,861] Making new env: MsPacman-v0



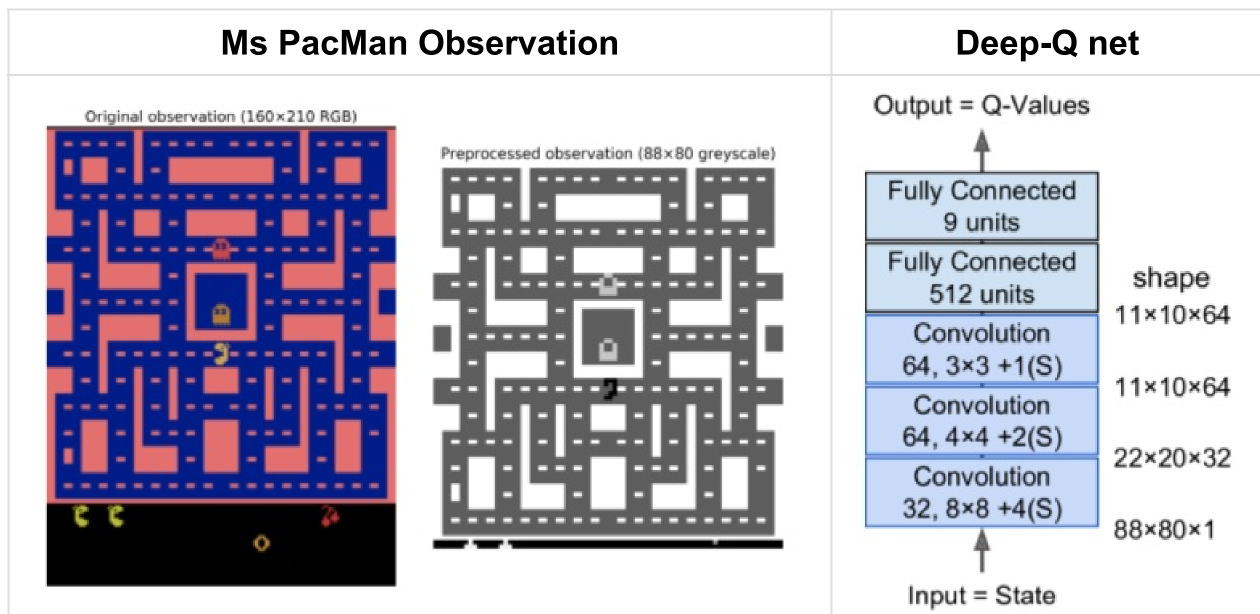


((210, 160, 3), Discrete(9))
```

```
mspacman_color = np.array([210, 164, 74]).mean()

# crop image, shrink to 88x80 pixels, convert to grayscale, improve contrast

def preprocess_observation(obs):
    img = obs[1:176:2, ::2] # crop and downsize
    img = img.mean(axis=2) # to greyscale
    img[img==mspacman_color] = 0 # improve contrast
    img = (img - 128) / 128 - 1 # normalize from -1. to 1.
    return img.reshape(88, 80, 1)
```




```
# Create DQN
# 3 convo layers, then 2 FC layers including output layer

from tensorflow.contrib.layers import convolution2d, fully_connected

input_height       = 88
input_width        = 80
input_channels     = 1
conv_n_maps        = [32, 64, 64]
conv_kernel_sizes = [(8,8), (4,4), (3,3)]
conv_strides       = [4, 2, 1]
conv_paddings      = ["SAME"]*3
```

```
conv_activation   = [tf.nn.relu]*3
n_hidden_in       = 64 * 11 * 10 # conv3 has 64 maps of 11x10 ea
ch
n_hidden          = 512
hidden_activation = tf.nn.relu
n_outputs         = env.action_space.n # 9 discrete actions are
available

initializer = tf.contrib.layers.variance_scaling_initializer()

# training will need ***TWO*** DQNs:
# one to train the actor
# another to learn from trials & errors (critic)
# q_network is our net builder.

def q_network(X_state, scope):
    prev_layer = X_state
    conv_layers = []

    with tf.variable_scope(scope) as scope:

        for n_maps, kernel_size, stride, padding, activation in
zip(
            conv_n_maps,
            conv_kernel_sizes,
            conv_strides,
            conv_paddings,
            conv_activation):

            prev_layer = convolution2d(
                prev_layer,
                num_outputs=n_maps,
                kernel_size=kernel_size,
                stride=stride,
                padding=padding,
                activation_fn=activation,
                weights_initializer=initializer)

            conv_layers.append(prev_layer)
```

```
        last_conv_layer_flat = tf.reshape(
            prev_layer,
            shape=[-1, n_hidden_in])

        hidden = fully_connected(
            last_conv_layer_flat,
            n_hidden,
            activation_fn=hidden_activation,
            weights_initializer=initializer)

        outputs = fully_connected(
            hidden,
            n_outputs,
            activation_fn=None,
            weights_initializer=initializer)

    trainable_vars = tf.get_collection(
        tf.GraphKeys.TRAINABLE_VARIABLES,
        scope=scope.name)

    trainable_vars_by_name = {var.name[len(scope.name):]: var
        for var in trainable_vars}

    return outputs, trainable_vars_by_name
```

```
# create input placeholders & two DQNs

X_state = tf.placeholder(
    tf.float32,
    shape=[None, input_height, input_width,
    input_channels])

actor_q_values, actor_vars   = q_network(X_state, scope="q_networks/actor")
critic_q_values, critic_vars = q_network(X_state, scope="q_networks/critic")

copy_ops = [actor_var.assign(critic_vars[var_name])
            for var_name, actor_var in actor_vars.items()]


# op to copy all trainable vars of critic DQN to actor DQN...
# use tf.group() to group all assignment ops together

copy_critic_to_actor = tf.group(*copy_ops)
```

```
# Critic DQN learns by matching Q-Value predictions
# to actor's Q-Value estimations during game play

# Actor will use a "replay memory" (5 tuples):
# state, action, next-state, reward, (0=over/1=continue)

# use normal supervised training ops
# occasionally copy critic DQN to actor DQN

# DQN normally returns one Q-Value for every poss. action
# only need Q-Value of action actually chosen
# So, convert action to one-hot vector [0...1...0], multiple by
Q-values
# then sum over 1st axis.

X_action = tf.placeholder(
    tf.int32, shape=[None])

q_value = tf.reduce_sum(
    critic_q_values * tf.one_hot(X_action, n_outputs),
    axis=1, keep_dims=True)
```

```
# training setup

tf.reset_default_graph()

y = tf.placeholder(
    tf.float32, shape=[None, 1])

cost = tf.reduce_mean(
    tf.square(y - q_value))

# non-trainable. minimize() op will manage incrementing it
global_step = tf.Variable(
    0,
    trainable=False,
    name='global_step')

optimizer = tf.train.AdamOptimizer(learning_rate)

training_op = optimizer.minimize(cost, global_step=global_step)

init = tf.global_variables_initializer()

saver = tf.train.Saver()
```

```
----------------------------------------------------------------
-----------

ValueError                                Traceback (most recent
 call last)

<ipython-input-54-ae5a849b8026> in <module>()
      7
      8 cost = tf.reduce_mean(
----> 9     tf.square(y - q_value))
     10
     11 # non-trainable. minimize() op will manage incrementing
it
```

```
/home/bjpcjp/anaconda3/lib/python3.5/site-packages/tensorflow/py
thon/ops/math_ops.py in binary_op_wrapper(x, y)
    879
    880   def binary_op_wrapper(x, y):
--> 881     with ops.name_scope(None, op_name, [x, y]) as name:
    882       if not isinstance(y, sparse_tensor.SparseTensor):
    883         y = ops.convert_to_tensor(y, dtype=x.dtype.base_
dtype, name="y")


/home/bjpcjp/anaconda3/lib/python3.5/contextlib.py in __enter__(
self)
     57     def __enter__(self):
     58         try:
---> 59             return next(self.gen)
     60         except StopIteration:
     61             raise RuntimeError("generator didn't yield")
 from None


/home/bjpcjp/anaconda3/lib/python3.5/site-packages/tensorflow/py
thon/framework/ops.py in name_scope(name, default_name, values)
   4217   if values is None:
   4218     values = []
-> 4219   g = _get_graph_from_inputs(values)
   4220   with g.as_default(), g.name_scope(n) as scope:
   4221     yield scope


/home/bjpcjp/anaconda3/lib/python3.5/site-packages/tensorflow/py
thon/framework/ops.py in _get_graph_from_inputs(op_input_list, g
raph)
   3966         graph = graph_element.graph
   3967       elif original_graph_element is not None:
-> 3968         _assert_same_graph(original_graph_element, graph
_element)
   3969       elif graph_element.graph is not graph:
   3970         raise ValueError(
```

```
/home/bjpcjp/anaconda3/lib/python3.5/site-packages/tensorflow/py
thon/framework/ops.py in _assert_same_graph(original_item, item)
   3905   if original_item.graph is not item.graph:
   3906     raise ValueError(
-> 3907         "%s must be from the same graph as %s." % (item,
 original_item))
   3908
   3909


ValueError: Tensor("Sum_1:0", shape=(?, 1), dtype=float32) must
be from the same graph as Tensor("Placeholder:0", shape=(?, 1),
dtype=float32).
```

```
# use a deque list to build the replay memory

from collections import deque

replay_memory_size = 10000
replay_memory = deque(
    [], maxlen=replay_memory_size)

def sample_memories(batch_size):
    indices = rnd.permutation(
        len(replay_memory))[:batch_size]
    cols = [[], [], [], [], []] # state, action, reward, next_st
ate, continue

    for idx in indices:
        memory = replay_memory[idx]
        for col, value in zip(cols, memory):
            col.append(value)

    cols = [np.array(col) for col in cols]
    return (cols[0], cols[1], cols[2].reshape(-1, 1), cols[3], c
ols[4].reshape(-1, 1))
```

```
# create an actor
# use epsilon-greedy policy
# gradually decrease epsilon from 1.0 to 0.05 across 50K training steps

eps_min = 0.05
eps_max = 1.0
eps_decay_steps = 50000

def epsilon_greedy(q_values, step):
    epsilon = max(eps_min, eps_max - (eps_max-eps_min) * step/eps_decay_steps)
    if rnd.rand() < epsilon:
        return rnd.randint(n_outputs) # random action
    else:
        return np.argmax(q_values) # optimal action
```

```
# training setup: the variables

n_steps = 100000 # total number of training steps
training_start = 1000 # start training after 1,000 game iterations
training_interval = 3 # run a training step every 3 game iterations
save_steps = 50 # save the model every 50 training steps
copy_steps = 25 # copy the critic to the actor every 25 training steps
discount_rate = 0.95
skip_start = 90 # skip the start of every game (it's just waiting time)
batch_size = 50
iteration = 0 # game iterations
checkpoint_path = "./my_dqn.ckpt"
done = True # env needs to be reset
```

```
# let's get busy
import os
```

```
with tf.Session() as sess:

    # restore models if checkpoint file exists
    if os.path.isfile(checkpoint_path):
        saver.restore(sess, checkpoint_path)

        # otherwise normally initialize variables
    else:
        init.run()

    while True:
        step = global_step.eval()
        if step >= n_steps:
            break

        # iteration = total number of game steps from beginning

        iteration += 1
        if done: # game over, start again
            obs = env.reset()

            for skip in range(skip_start): # skip the start of each game
                obs, reward, done, info = env.step(0)
            state = preprocess_observation(obs)

        # Actor evaluates what to do
        q_values = actor_q_values.eval(feed_dict={X_state: [state]})
        action   = epsilon_greedy(q_values, step)

        # Actor plays
        obs, reward, done, info = env.step(action)
        next_state = preprocess_observation(obs)

        # Let's memorize what just happened
        replay_memory.append((state, action, reward, next_state, 1.0 - done))
        state = next_state
```

```
        if iteration < training_start or iteration % training_in
terval != 0:
            continue

        # Critic learns
        X_state_val, X_action_val, rewards, X_next_state_val, co
ntinues = (
            sample_memories(batch_size))

        next_q_values = actor_q_values.eval(
            feed_dict={X_state: X_next_state_val})

        max_next_q_values = np.max(
            next_q_values, axis=1, keepdims=True)

        y_val = rewards + continues * discount_rate * max_next_q
_values

        training_op.run(
            feed_dict={X_state: X_state_val, X_action: X_action_
val, y: y_val})

        # Regularly copy critic to actor
        if step % copy_steps == 0:
            copy_critic_to_actor.run()

        # And save regularly
        if step % save_steps == 0:
            saver.save(sess, checkpoint_path)

        print("\n",np.average(y_val))
```

```
 1.09000234097

 1.35392784142

 1.56906713688
```

```
2.5765440191

1.57079289043

1.75170834792

1.97005553639

1.97246688247

2.16126081383

1.550295331

1.75750140131

1.56052656734

1.7519523176

1.74495741558

1.95223849511

1.35289915931

1.56913152564

2.96387254691

1.76067311585

1.35536773229

1.54768545294

1.53594982147

1.56104325151
```

```
1.96987313104

2.35546155441

1.5688166486

3.08286282682

3.28864161086

3.2878398273

3.09510449028

3.09807873964

3.90697311211

3.07757974195

3.09214673901

3.28402029777

3.28337000942

3.4255889504

3.49763186431

2.85764229989

3.04482784653

2.68228099513

3.28635532999

3.29647485089
```

```
3.07898310328

3.10530596256

3.27691918874

3.09561720395

2.67830030346

3.09576807404

3.288335078

3.0956065948

5.21222548962

4.21721751595

4.7905973649

4.59864345837

4.39875211382

4.51839643717

4.59503188992

5.01186150789

4.77968219852

4.78787856865

4.20382899523

4.20432999897
```

```
5.0028930707

5.20069698572

4.80375980473

5.19750945711

4.20367767668

4.19593407536

4.40061367989

4.6054182477

4.79921974087

4.38844807434

4.20397897291

4.60095557356

4.59488785553

5.75924422598

5.75949315596

5.16320213652

5.36019721937

5.56076610899

5.16949163198

5.75895399189
```

```
 5.96050115204

 5.97032629395




---------------------------------------------------------------
-----------

KeyboardInterrupt                         Traceback (most recent
 call last)

<ipython-input-44-d0da605267f3> in <module>()
     46
     47         next_q_values = actor_q_values.eval(
---> 48             feed_dict={X_state: X_next_state_val})
     49
     50         max_next_q_values = np.max(


/home/bjpcjp/anaconda3/lib/python3.5/site-packages/tensorflow/py
thon/framework/ops.py in eval(self, feed_dict, session)
    579
    580     """
--> 581     return _eval_using_default_session(self, feed_dict,
self.graph, session)
    582
    583


/home/bjpcjp/anaconda3/lib/python3.5/site-packages/tensorflow/py
thon/framework/ops.py in _eval_using_default_session(tensors, fe
ed_dict, graph, session)
   3795                         "the tensor's graph is different
from the session's "
   3796                         "graph.")
-> 3797   return session.run(tensors, feed_dict)
   3798
   3799
```

```
/home/bjpcjp/anaconda3/lib/python3.5/site-packages/tensorflow/py
thon/client/session.py in run(self, fetches, feed_dict, options,
 run_metadata)
    765     try:
    766       result = self._run(None, fetches, feed_dict, optio
ns_ptr,
--> 767                          run_metadata_ptr)
    768       if run_metadata:
    769         proto_data = tf_session.TF_GetBuffer(run_metadat
a_ptr)


/home/bjpcjp/anaconda3/lib/python3.5/site-packages/tensorflow/py
thon/client/session.py in _run(self, handle, fetches, feed_dict,
 options, run_metadata)
    963     if final_fetches or final_targets:
    964       results = self._do_run(handle, final_targets, fina
l_fetches,
--> 965                              feed_dict_string, options,
run_metadata)
    966     else:
    967       results = []


/home/bjpcjp/anaconda3/lib/python3.5/site-packages/tensorflow/py
thon/client/session.py in _do_run(self, handle, target_list, fet
ch_list, feed_dict, options, run_metadata)
   1013     if handle is None:
   1014       return self._do_call(_run_fn, self._session, feed_
dict, fetch_list,
-> 1015                            target_list, options, run_met
adata)
   1016     else:
   1017       return self._do_call(_prun_fn, self._session, hand
le, feed_dict,


/home/bjpcjp/anaconda3/lib/python3.5/site-packages/tensorflow/py
thon/client/session.py in _do_call(self, fn, *args)
```

```
   1020   def _do_call(self, fn, *args):
   1021     try:
-> 1022       return fn(*args)
   1023     except errors.OpError as e:
   1024       message = compat.as_text(e.message)


/home/bjpcjp/anaconda3/lib/python3.5/site-packages/tensorflow/py
thon/client/session.py in _run_fn(session, feed_dict, fetch_list
, target_list, options, run_metadata)
   1002         return tf_session.TF_Run(session, options,
   1003                                  feed_dict, fetch_list,
target_list,
-> 1004                                  status, run_metadata)
   1005
   1006     def _prun_fn(session, handle, feed_dict, fetch_list)
:


KeyboardInterrupt:
```